Note del corso[1]:

COMPUTING METHODS FOR EXPERIMENTAL PHYSICS AND DATA ANALYSIS

Compilato il October 30, 2022

Lorenzo Zaffina

Università di Pisa

[1]Queste note nascono dalle lezioni del corso CMEPDA dell'a.a. 2022-23. Per realizzarle ho in gran parte utilizzato il materiale disponibile su github al link `https://github.com/lucabaldini/cmepda.git`.

Contents

Chapter 1

Modulo Base - Scientifc Python

Lun 20 sett - Lezione 1

Lecture basic 1: Development workflow

L'importanza della riproducibilità

La ricerca scientifica dovrebbe essere prima di tutto *corretta* e *riproducibile*. In particolare, *riproducibile* vuol dire che deve essere possibile, partendo dagli stessi dati, arrivare alle stesse conclusioni. Il software ricopre un ruolo fondamentale nella moderna fisica sperimentale. Per questo motivo deve essere trattato alla stregua di un esperimento scientifico, cioè deve essere sviluppato in maniera controllata e deve essere riproducibile.

Version control

"Version control": *A component of software configuration management, version control, also known as revision control or source control, is the management of changes to documents, computer programs, large web sites, and other collections of information*[1]

Un sistema di Version Control permette di raccogliere metadati[2] sul codice ogni qual volta il codice subisce delle modifiche.

Terminologia

- **Repository**: the place where files' current and historical data are stored
- **Revision or version**: the state at a point in time of the entire tree in the repository
- **Clone**: creating a repository containing the revisions from another repository
- **Working copy**: a local copy of files from a repository at a specific revision
- **Checkout**: create a local working copy from the repository
- **Change or diff**: a specific modification to a set of files under version control
- **Commit**: write the changes made in the working copy back to the repository

Tipologie di Version Control System

Nel corso degli anni sono state sviluppate diverse tipologie di sistemi di controllo di configurazione. I principali sono i seguenti:

[1] Wikipedia

[2] In informatica il metadato è un sistema strutturato di dati sui dati. Il suo scopo è di descrivere il contenuto, la struttura e l'ambito in cui s'inquadra un documento informatico, per la sua gestione e archiviazione nel tempo.

Local version control systems (e.g. RCS)

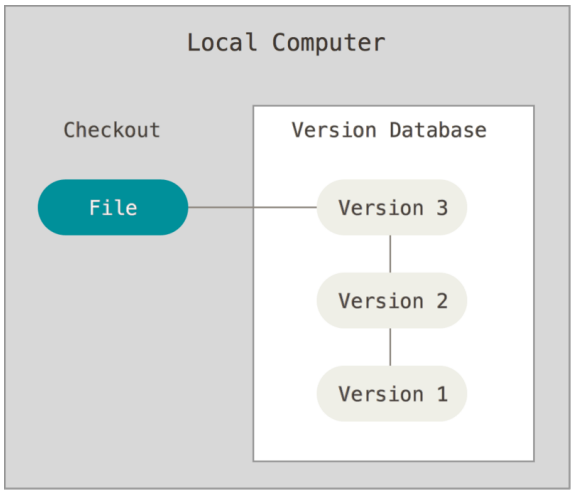


Figure 1.1: Local version control system: Keeps differences between revisions in a local database. Can recreate what any file looked like at any point in time.

Centralized Version Control Systems (e.g. CVS, Subversion)

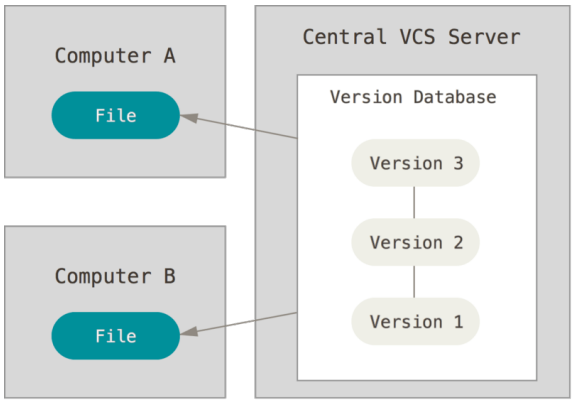


Figure 1.2: Centralized Version Control System: Single server containing all the versioned files. Clients can check out the files from the repository. Most popular model through most of the '90.

Il work-flow tipico dei Centralized Version Control System è molto basico:

1. Check out a local working copy from the remote server
2. Modify the working copy
3. Commit the changes back to the repository

Distributed version control system (e.g. git, mercurial)

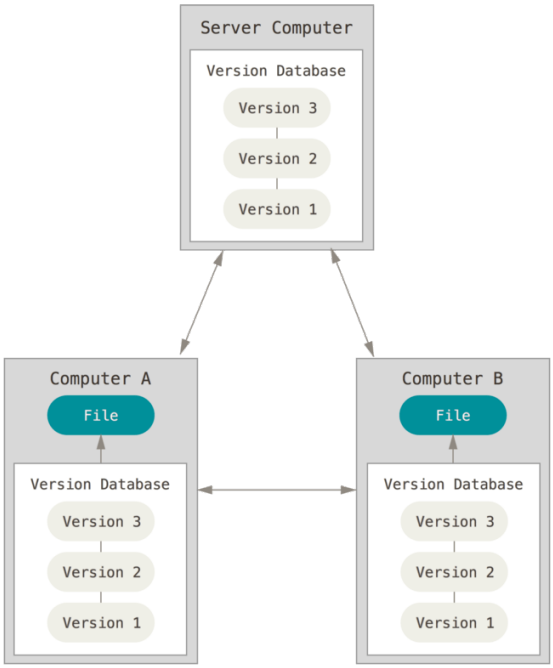


Figure 1.3: Distributed version control system: Clients fully mirror the repository, including its full history. Allows for a much richer variety of work-flows.

Versioning single files vs. the entire repository

Old VCS only tracked modification on a file-by-file basis; i.e., CVS assigns revision numbers to the single files. All modern VCS track a whole commit as a new revision; i.e., revisions are assigned to the repository.
It makes a lot of sense to version the entire repository. Versioning single files can give a cozy feeling, but when files interact with each other you need to know the status of the entire repository to reliably predict the output!

Centralized vs. distributed VCS

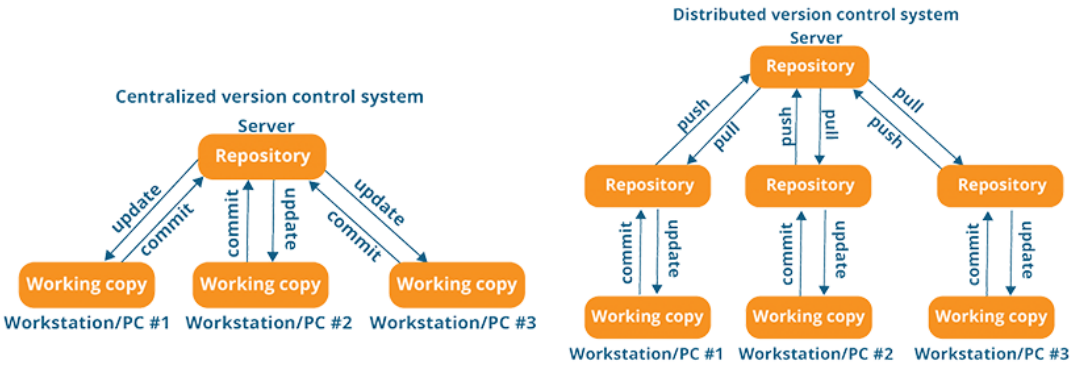


Figure 1.4: Confronto tra un sistema di controllo centralizzato ed uno distribuito.

Il workflow nel caso del *centralized VCS* è **lineare**: *Subversion assigns a progressive number to the repo at each commit.*

Invece nel caso di un *Distributed VCS*, il workflow è intrinsecamente **non lineare**: *Any one given local repository is not ahead or behind any other repository—just different.*
Ma allora come facciamo ad assegnare una versione (revision) in un sistema distribuito? Per capirlo facciamo una piccola digressione sulle funzioni di hash:

Funzioni di Hash

Nel linguaggio matematico e informatico, l'hash è una funzione non invertibile che mappa una stringa di lunghezza arbitraria in una stringa di lunghezza predefinita. Esistono numerosi algoritmi che realizzano funzioni hash con particolari proprietà che dipendono dall'applicazione.[3]

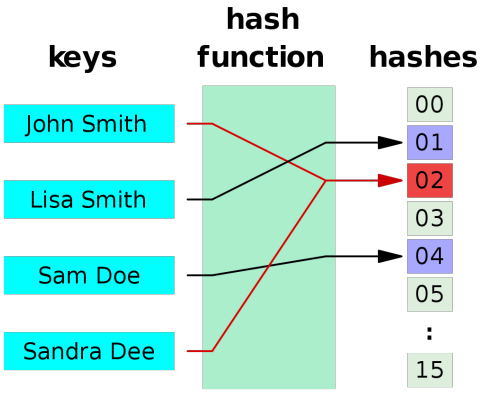


Figure 1.5: Esempio di funzione di hash.

Hash function maps data of arbitrary size to fixed-size values (e.g., anything to an integer).

Alcune proprietà che deve avere una buona funzione di hash:

- Deterministica.
- Uniforme nello spazio immagine (minimicca le collisioni).
- Facile da calcolare (e, possibilmente, difficile da invertire).

```
1  print(hash(3))
2  print(hash(3.))
3  print(hash(3.001))
4  print(hash('hello'))
5  print(hash('Hello'))
6
7  [Output]
8  3
9  3
10 2305843009213443
11 -8080512805622017032
12 -8706679013462221575
```

Every immutable object is hashable in Python. Each type gets its own algorithm. This is important because in distributed VCS each commit gets its own hash.

Altra Terminologia

[3] Wikipedia

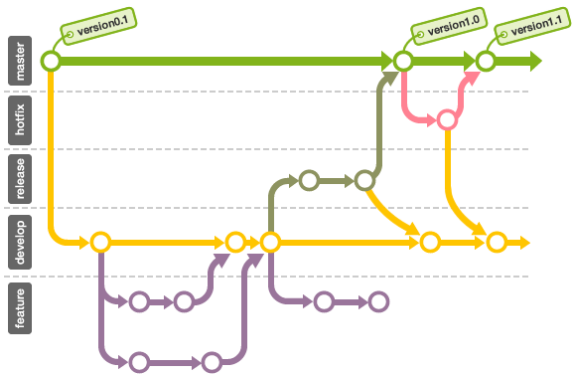


Figure 1.6: Esempio di branching.

- **Branches:** alternative paths where more copies of the same files develop in different ways independently.
- **Master, or trunk, or tip:** the unique line of development that is not a branch.
- **Merge:** application of two sets of changes to a set of files.
- **Conflict:** changes to the same file by two or more developers that the system is unable to reconcile.

Gio 22 sett - Lezione 2

Lecture basic 2: Python Basics (1/2)

PEP: Python Enhancement Proposal

"PEP stands for Python Enhancement Proposal, and there are several of them. A PEP is a document that describes new features proposed for Python and documents aspects of Python, like design and style, for the community."

Coding Conventions

Ci sono delle linee guida su come scrivere il codice. Queste si chiamano *Coding Conventions*, e differiscono per ogni linguaggio.
In particolare la **PEP8**[4] codifica la coding convention in python.

Un esempio può essere quello di usare per le indentazioni gli spazi anziché i tab. Questo perché il tab dipende dall'editor di testo usato e questo è un male!

Esistono dei tool che permettono di controllare automaticamente quanto un codice è *Pythonico*, ad esempio **pylint**[5]. É buona abitudine, prima di pushare un file su github, usare pylint per controllare che non ci siano errori nel codice!.

Variables and basic types

Python è quello che si chiama *linguaggio tipizzato forte*[6]. Nonostante ciò, in python le variabili non si dichiarano. Posso scrivere:

```
1 x = 3
2 x = 'ciao'
```

senza che mi dia errore.
Invece in C (così come nella maggior parte dei linguaggi compilati), una volta dichiarata una variabile, essa è di quel tipo per sempre!

Vediamo ora le principali strutture dati di python. Ricordiamo che le strutture dati si differenziano in mutabili e immutabili. In particolare, le strutture dati sono definite dalle operazioni che ci posso fare sopra.

- **Numeri** (Interi, Float): notiamo che in python esiste un solo tipo di interi, detto "a precisione illimitata": posso rappesentare un numero arbitrariamente grande finché non finisco la RAM.

- **Stringhe**: è equivalente usare le virgolette singole (' ') o doppie (" ").

- **Liste**: le liste sono sequenze di elementi, che possono essere di qualsiasi tipo, anche liste stesse. Ad esempio posso accedere al primo elemento della lista `l` facendo `l[0]`; oppure cambiare un elemento facendo `l[0]='ciao'`. Infine posso appendere un elemento alla fine facendo `l.append('last')`.

- **Tuple**: ad esempio `t = (1,2,3)`. Una tupla è come una lista, ma è immutabile. Posso sempre accedere al primo elemento della tupla `t` facendo `t[0]`, ma **non** posso cambiare un elemento facendo ad esempio `l[0]=33`.
  Se scrivo su terminale: `(1,2,3)+(5,6,7)`, otterrò la nuova tupla `(1,2,3,5,6,7)`.

[4] dargli un'occhiata `https://peps.python.org/pep-0008/`
[5] `https://www.pylint.org/`
[6] In un linguaggio fortemente tipizzato, il programmatore è tenuto a specificare il tipo di ogni elemento sintattico che durante l'esecuzione denota un valore (per esempio un valore costante, una variabile o un'espressione), e il linguaggio garantisce che tale valore sia utilizzato in modo coerente con il tipo specificato: per esempio, non è possibile eseguire una somma aritmetica su dati di tipo stringa. Questo concetto generale può applicarsi con diverse sfumature; a seconda del contesto.

- **Dizionari**: Sono oggetti di tipo chiave-valore, ovvero mappano (mediante hash table o albero binario) una chiave in un valore. Sono efficienti nel trovare il valore associato alla chiave. Ad esempio `'a':3, 'b':4`.

Di seguito alcuni esempi:

```
i = 3
x = 3.0
print(i, type(i))
print(x, type(x))
s = 'Hi there!'
print(s, type(s))
l = [1, 2, 'a string']
print(l, type(l), l[0])
t = (1, 2, 'a string')
print(t, type(t), t[0])
d = {'key1': 1, 'key2': 2}
print(d, type(d), d['key1'])
[Output]
3 <class 'int'>
3.0 <class 'float'>
Hi there! <class 'str'>
[1, 2, 'a string'] <class 'list'> 1
(1, 2, 'a string') <class 'tuple'> 1
'key1': 1, 'key2': 2 <class 'dict'> 1
```

String Formatting

É poco pithonico usare l'operatore + per unire due stringhe, facendo ad esempio `'Luca' + 'Baldini'`.
É anche preferibile evitare usare l'operatore %.
La cosa migliore è usare le *f-string*, come si vede nel seguente esempio:

```
name = 'Luca'
age = 42

# The ugly way.
print('My name is ' + name + ' I am ' + str(age) + ' year(s) old.')

# The old way (% operator)
print('My name is %s I am %d year(s) old.' % (name, age))

# The new way (.format)
# This is actually *much* more powerful and flexible than implied here.
print('My name is {} I am {} year(s) old.'.format(name, age))

# The newer way---new in Python 3.6. This is awesome!
print(f'My name is {name} I am {age} year(s) old.')


[Output]
My name is Luca I am 42 year(s) old.
My name is Luca I am 42 year(s) old.
My name is Luca I am 42 year(s) old.
My name is Luca I am 42 year(s) old.
```

Le Funzioni

DRY (Don't Repeat Yourself) is better than WET (Write Every Time). É importante dare un nome chiaro ed esplicativo alle funzioni che scriviamo.

```
import math
def square(x):
"""Return the square of x.
"""
return x * x
def cartesian_to_polar(x=1., y=1.):
"""Convert cartesian to polar coordinates.
"""
r = math.sqrt(x**2. + y**2.)
phi = math.atan2(y, x)
return r, phi
print(square(2.))
print(cartesian_to_polar(0., 1.))
print(cartesian_to_polar())
[Output]
4.0
(1.0, 1.5707963267948966)
(1.4142135623730951, 0.7853981633974483)
```

Funzioni Variadiche

Sono delle funzioni che accettano un numero variabile di argomenti. Ad esempio:

```
import os
p1 = os.path.join('path', 'to', 'my', 'file')
p2 = os.path.join('howdy', 'partner')
print(p1)
print(p2)
s1 = sum([1, 2])
s2 = sum([1, 2, 3, 4, 5])
print(s1)
print(s2)
[Output]
path/to/my/file
howdy/partner
3
15
```

Arbitrary argument lists

```
import os
def join1(*args):
"""Horrible: do not use the + operator with strings in a loop.
"""
out = ''
for arg in args:
out += '%s/' % arg
return out.rstrip('/')
def join2(*args):
"""This a more sensible version---and you get the idea of the *.
"""
return '/'.join(args)
def join3(*args, sep=os.path.sep):
"""Even better---this will work on any OS.
"""
return sep.join(args)
print(join1('path', 'to', 'file'))
print(join2('path', 'to', 'file'))
print(join3('path', 'to', 'file'))
```

```
[Output]
path/to/file
path/to/file
path/to/file
```

Un esempio: la funzione di fit

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
x = np.linspace(0., 10., 11)
y = 2.5 + 3.2 * x
def model(x, m, q):
return m * x + q
popt, pcov = curve_fit(model, x, y)
plt.errorbar(x, y, fmt='o')
# Overlay the model without unpacking the best-fit parameters.
plt.plot(x, model(x, *popt))
# Compare with
# mhat, qhat = popt
# plt.plot(x, model(x, mhat, qhat))
```

Keyword arguments

Keyword arguments (or named arguments) are values that, when passed into a function, are identifiable by specific parameter names. A keyword argument is preceded by a parameter and the assignment operator, = . Keyword arguments can be likened to dictionaries in that they map a value to a keyword.

```
def func(**kwargs):
"""
"""
print(kwargs.get('verbose', False))
func()
func(verbose=True)
func(verbose=False)
func(verbose=True, num_events=3)
func(True)
[Output]
False
True
False
True
Traceback (most recent call last):
File "snippets/func_kwargs.py", line 11, in <module>
func(True)
TypeError: func() takes 0 positional arguments but 1 was given
```

Basic control flow

```
i = 2
# Conditional expressions
if i == 2:
print('Apple')
elif i == 3:
print('Peach')
else:
print('Cheese')
# For loops
```

```
for i in [1, 2, 3]:
print(i)
# While loops
while i != 0:
print(i)
i -= 1
[Output]
Apple
1
2
3
3
2
```

Advanced Iteration

```
list1 = ['a', 'b', 'c']
list2 = [10, 11, 12]
# Horrible (and very un-Pythonic, too)!
for i in range(len(list1)):
print(i, list1[i])
# Nice-looking.
for i, item in enumerate(list1):
print(i, item)
# Zipping iterables
for item1, item2 in zip(list1, list2):
print(item1, item2)
# List comprehension
print([x**2 for x in list2])
[Output]
0 a
1 b
2 c
0 a
1 b
2 c
a 10
b 11
c 12
[100, 121, 144]
```

Nota: I numeri in virgola mobile sono esatti

Consideriamo il seguente esempio:

```
[lbaldini@nbbaldini slides]$ python

Python 3.7.4 (default, Jul 9 2019, 16:32:37)
[GCC 9.1.1 20190503 (Red Hat 9.1.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 0.1 + 0.2 == 0.3
False
>>> 0.2 + 0.2 == 0.4
True
```

Cosa sta succedendo? Il fatto è che, essendo inesatti, non ha senso chiedere se due numeri in virgola mobile siano esatti!
Quando scriviamo un numero in virgola mobile, per il PC è sempre un numero raionale, in quanto è troncato.

Rappresentazione in virgola mobile

[...]

References

- https://scipy-lectures.org/
- https://docs.quantifiedcode.com/python-anti-patterns/
- https://sebastianraschka.com/Articles/2014_python_2_3_key_diff.html
- https://www.python.org/dev/peps/pep-0020/
- https://www.python.org/dev/peps/pep-0008/
- https://docs.python-guide.org/writing/style/
- https://docs.python.org/3/library/stdtypes.html
- https://docs.python.org/3/tutorial/controlflow.html#defining-functions
- https://docs.python.org/3/tutorial/floatingpoint.html
- https://floating-point-gui.de/
- https://www.itu.dk/~sestoft/bachelor/IEEE754_article.pdf

Lecture basic 3: Python Basics (2/2)

La Python Standard Lybrary

- La gerarchia è sostanzialmtnete la seguente:
  - The Python core language (all you get at the interpreter startup)
  - The Python standard library (e.g., `math`)
  - An enourmous number of third-party packages (e.g., `numpy`)
  - Eventuali librerie scritte da noi
- The standard library is included in every Python distribution
  - And it is (slowly) evolving with time
- With third-party packages you are on your own
  - Although Anaconda solves many of the issues
  - And if you are using GNU-Linux your package manager is probably taking care of everything for you
- (Well—and of course there are your own modules, too...)
- Anything that is out of the core is loaded in memory via an `import` statement

Il sistema di Import

```
from math import *
[...]
# Terrible: where the hell is sqrt coming from?
x = sqrt(2.)
from math import sqrt
[...]
# Better: if you haven't redefined sqrt this is from the math library
x = sqrt(2.)
import math
[...]
# Best: five more characters, but at least is clear where sqrt is coming from
x = math.sqrt(2.)
```

- The `$PYTHONPATH` environmental variables is your friend to control where you want to import modules from
  - You will need to tweak it when you start writing your own packages
- You will need suitable `__init__.py` files to navigate directories

Nota: non abusare del sistema di import! Di seguito un esempio ok:

```
# This is ok, and vastly recognized by the community
import numpy as np
from matplotlib import pyplot as plt
x = np.linspace(0., 10., 100)
y = x**2.
plt.plot(x, y)
```

Mentre il seguente esempio è una catastrofe!

```
from math import *
import logging as log
# ... 1000 lines of code in the middle
x = log(2.)
[Output]
Traceback (most recent call last):
File "snippets/import2.py", line 6, in <module>
x = log(2.)
TypeError: 'module' object is not callable
```

La Standard Library: `time`, `datetime` and `calendar`

- Collections of facilities related to date and time
  - Measure the execution time of your scripts
  - Convert from time to date and vice-versa

"Il tempo è una cosa seria" -Luca Baldini.

La Standard Library: `math`

```
Python 3.7.4 (default, Jul 9 2019, 16:32:37)
[GCC 9.1.1 20190503 (Red Hat 9.1.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import math
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>>
```

Nota: lavorando molto con gli array, ci troveremo ad usare principalmente `numpy` e `scipy`.

La Standard Library: `random`

```
Python 3.7.4 (default, Jul 9 2019, 16:32:37)
[GCC 9.1.1 20190503 (Red Hat 9.1.1-1)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import random
>>> print(dir(random))
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
'SystemRandom', 'TWOPI', '_BuiltinMethodType', '_MethodType', '_Sequence',
'_Set', '__all__', '__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__spec__', '_acos', '_bisect', '_ceil',
'_cos', '_e', '_exp', '_inst', '_itertools', '_log', '_os', '_pi', '_random',
'_sha512', '_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn',
'betavariate', 'choice', 'choices', 'expovariate', 'gammavariate', 'gauss',
'getrandbits', 'getstate', 'lognormvariate', 'normalvariate', 'paretovariate',
'randint', 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle',
'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']
>>>
```

Anche qui, useremo principalmente `numpy`.

La Standard Library: `os`, `os.path`, `glob` and `shutil`

Servono ad interagire con il sistema operativo:

- Miscellaneous operating system interfaces
  - Access filesystem (access, create and copy files and directories)
  - List directory content
  - Environmental variables
  - Absolute and relative paths
  - Exec OS commands
- All of this in a cross-platform fashion!

La Standard Library: `argparse`

É utilissimo per passare informazioni direttamente dalla linea di comando.

- Ever found yourself modifying the source code and running your program with different parameters?
  - This is a terribly bad practice!
  - And git will complain about modified files :-)
- Keep the argparse documentation under your pillow!

La Standard Library: `logging`

- Ever found yourself inserting debug print() statements in the code when needed?
  - This is another terrible bad practice!
  - And git will complain about modified files :-)
- Imagine if there was a thing that:
  - allowed to label messages with different levels of severity (e.g., debug, info, warning, error)
  - dynamically set a global filter on the severity level (e.g., do not print debug messages)
- This thing exists and is called `logging`
- Always prefer `logging` over `print`

Esiste anche un altro modulo molto usato che si chiama `Loguru`. Permette anche di stampare i log su un **logfile**.

Typical layout of a Python package

Say you have a project called sample:

```
README.rst
LICENSE
setup.py
requirements.txt
sample/__init__.py
sample/core.py
sample/helpers.py
docs/conf.py
docs/index.rst
tests/test_basic.py
tests/test_advanced.py
```

- Here is how the repository layout might look like:
  - README.rst
  - LICENSE (when in doubt use GPL v3)
  - requirements.txt (dependencies, for pip)
  - sample (actual python code, note it's the same name as the project)
  - docs (documentation)
  - tests (unit tests)
- We shall talk a lot about installation, documentation and unit tests in the second part of the course (advanced Python)

References

- https://docs.python.org/3/library/
- https://pypi.org/
- https://docs.python.org/3/reference/import.html
- https://docs.python-guide.org/
- https://docs.quantifiedcode.com/python-anti-patterns/

Gio 29 sett - Lezione 3

Lecture basic 4: Algorithms and data structures

Un algoritmo è una sequenza di istruzioni che dicono in modo **non ambiguo** come risolvere un problema. Usare un algoritmo piuttosto che un altro può comportare una grande differenza in termini di efficienza di tempi.

- Algorithms can be expressed in several different ways:
  - Flowcarts
  - Pseudo-code
  - Working code snippets
- The sequence of operation must be expressed **unambigously**

Esempio: ricerca sequenziale vs ricerca binaria

Problema: trovare un elemento in una lista ordinata.
Posso fare 2 cose:
1. Forza bruta: Faccio un loop sulla lista finché non trovo (o no) l'elemento cercato. Questo diventa sempre più sconveniente man mano che la lista si allunga (se la lista è lunga N, in media dovrò controllare N/2 elementi).

2. Ricerca Binaria:

- Start from the middle (if that's the target you're done)
- If the target is smaller (larger) than the element in the center, bisect the half-list on the left (right)
- Iterate until you've found the target (or exhausted the list)

In questo caso dovrò controllare in media $\log_2(n)$ elementi. **Il logaritmo fa una bella differenza!**

Complessità di un algoritmo

Misura del costo computazionale di un algoritmo. Lo quantifico come funzione della grandezza dell'input che noi diamo all'algoritmo. Per es se il nostro algoritmo agisce su una lista, è interessante vedere come scala il tempo di operazione in funzione della lunghezza della lista.
Ordine di grandezza del numero di istruzioni elementari è una buona stima del tempo di esecuzione del programma. (anche se le istruzioni elementari possono durare un po' diversamente da pc a pc e da linguaggio a linguaggio).
Il numero di istruzioni fondamentali che un algoritmo esegue dipende dai dati che gli diamo in ingresso (ad es, a parità di lunghezza della lista, dipende da come è ordinata la lista stessa). Posso comunque chiedermi cosa succede nel *best case*, nel *worst case* e in *average*.

```
1  def find_maximum(list_):
2  """Find the biggest element in a list.
3  """
4  maximum = list_[0]
5  for value in list_[1:]:
6  if value > maximum:
7  maximum = value
8  return maximum
9
10 l = [1, 2, 5, 98, 3, 1672, 6, 34, 651]
11 print(find_maximum(l))
12
13 [Output]
14 1672
```

- Example: find the largest element in a list of length n
- How many fundamental instructions is this code executing?
  - (You should realize this is an ill-posed question)
  - One assignment and one list lookup at the beginning: 2
  - One lookup, one assignment and one comparison for each iteration in the for loop: $3(n-1)$
  - A variable number of assignments: between 0 and $(n-1)$
  - One final return instruction: 1
- Answer: anything between $3n$ and $4n-1$
  - (Depending on the input list)
- Message #1: the exact number of fundamental instructions that an algorithm performs is not determined a priori
  - It depends on the input data, instead
  - And so does the running time
- There's a few questions that you can legitimetely ask, anyway
  - How many instructions in the worst case?
  - How many instructions in the best case?
  - How many instructions on average?
- Message #2: the *exact* number of operations doesn't really matter, does it?
  - Different machines have different executions speed
  - Different languages have different meaning of *fundamental instruction*
- Message #3: still, the running time is related to the number of fundamental instructions

Andamento asintotico e notazioni O-grandi

- Say you have an algorithm operating on an input of lenght n
  - e.g., a list with n elements
  - or a string with n characters
- How many fundamental instructions N does it take to for your algorithm to run?

$$N = f(n)$$

- Asymptotic behavior: drop all the terms that grow slowly with n and only keep the one that grows faster

$$4n-1 \approx 4n \quad \text{and} \quad 2n^2+6n+3 \approx 2n^2$$

  (for large n)
- Let's go one step further, and say that we neglect the multiplicative factor in front of the leading term (posso ignorare il fattore moltiplicativo perché tanto non c'è una corrispondenza 1 ad 1 tra il numero di op elementari e il tempo).

$$4n-1 \approx n \quad \text{and} \quad 2n^2+6n+3 \approx n^2$$

- big-O notation: the two algorithms are $O(n)$ and $O(n^2)$

NB: Se un algoritmo ha complessità n^2 se ho un input 10 volte più grande, il tempo impiegato sarà 100 volte più grande. (**un algoritmo di complessità n^2 fa schifo**)

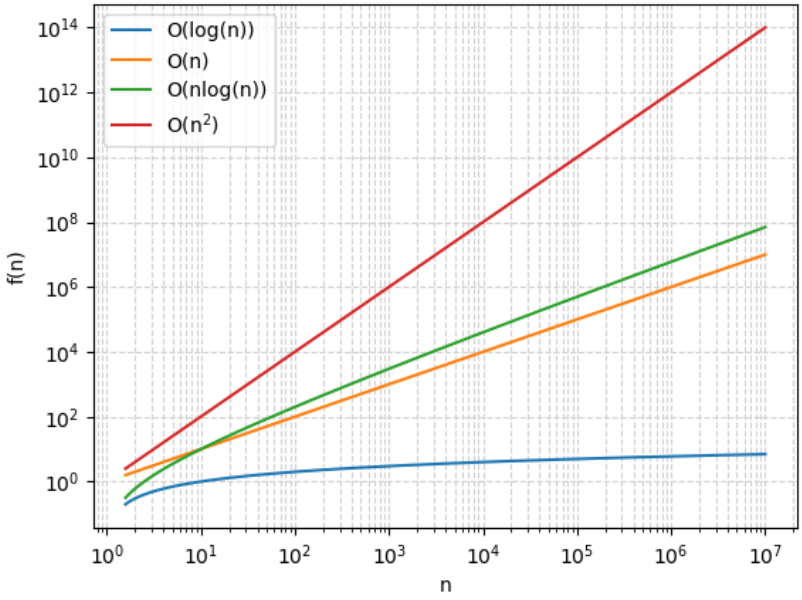


Figure 1.7: Andamenti asintotici di vario tipo. If $n = 10^6$ and you can beat down the complexity from n^2 to $n \log(n)$ you are cutting down the execution time by one million!

Come misuro il comportamento asintotico?

Soprattutto in casi in cui ci sono un sacco di linee di codice.

- **Forza Bruta**
  - Implement the algorithm
  - Run it on input data of different size and time the run
  - (Be careful: results may vary from run to run)
  - Plot the running time vs. input size
- **Per Analisi**:
  - Go ahead and count the instructions
  - Evaluate the best, worst and average case
  - (This can be difficult for complex programs, and subject to the hidiosyncrasis of the language)
- **Ad Occhio**
  - un loop ha tipicamente un costo O(n)
  - anche due loop consecutivi hanno costo O(n)
  - due loop annidati hanno un costo O(n^2) appena vedo due loop annidati è un cattivo segno! A volte non si possono evitare, a volte sì!

Una lettura leggera sulla "Complexity of Songs": `http://www.cs.bme.hu/~friedl/alg/knuth_song_complexity.pdf`

Strutture dati: le liste

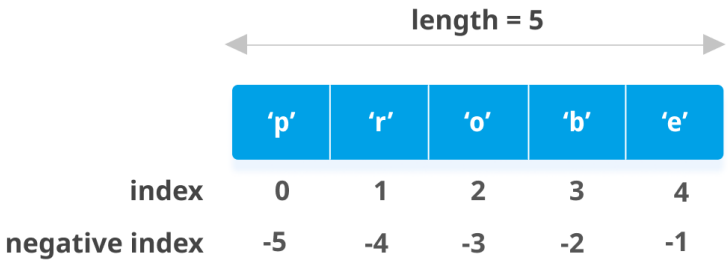


Figure 1.8

Operation	Average case	Worst case
Copy	O(n)	O(n)
Append	O(1)	O(1)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
min(s), max(s)	O(n)	
Get Length	O(1)	O(1)

Quando tolgo un elemento, una volta tolto devo poi rispostare tutto il resto! Per questo ho O(n).

Hash table

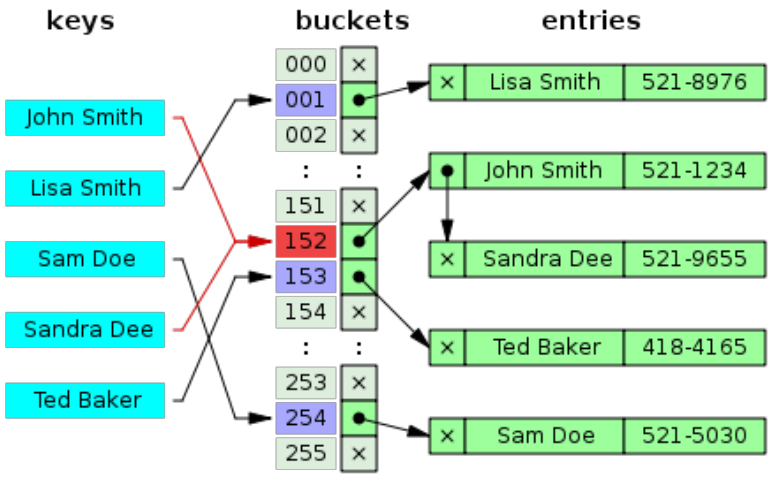


Figure 1.9: Hash table

- Associative array mapping keys to values
- Basic idea:
  - Pre-allocate some space (which might grow or shrink)
  - Keys are mapped to indices via a hash function
  - This is about it, except that you have to be able to handle collisions
- Hash tables (aka dictionaries) are highly optimized in Python

Strutture dati: I dizionari

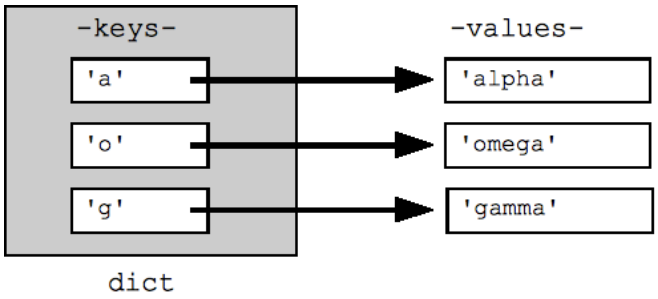


Figure 1.10: Dizionario

Operation	Average case	Worst case
Copy	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration	O(n)	O(n)

Per prendere un elemento, se non ci sono conflitti mi basta calcolare la funzine di hash sulla chiave. Ma se ci sono dei conflitti, nel caso peggiore in cui la struttura è piena, avrò n conflitti O(n).
I dizionari brillano nell'inserzione e nella cancellazione. I dizionari sono ottimi per i contesti in cui devo spesso inserire e/o cancellare cose.

```
print(hash(3))
print(hash(3.))
d = {}
d[3] = 'Hi there!'
print(d)
d[3.] = 'How are you?'
print(d)

[Output]
3
3
3: 'Hi there!'
3: 'How are you?'
```

- When a float corresponds to an integer, its hash is the same as that of the integer
- The hash is used to map keys into indices
- Therefore: 3 and 3. are the same key to a dictionary

Sorting

Programma che data una lista di numeri in virgola mobile, mi restituisce un'altra lista con gli stessi elementi, ma in ordine.

```
def sloppy_sort(list_):
"""Poor man's implementation of a sorting algorithm.
"""
sorted_list = []
for item in list_:
if len(sorted_list) == 0:
```

```
sorted_list.append(item)
else:
if item < sorted_list[0]:
sorted_list.insert(0, item)
else:
for i, sorted_item in enumerate(sorted_list):
if item <= sorted_item:
sorted_list.insert(i, item)
break
return sorted_list
l = [10, 1, 5, 2, 7, 3, 9, 4]
print(l)
print(sloppy_sort(l))
[Output]
[10, 1, 5, 2, 7, 3, 9, 4]
[1, 2, 3, 4, 5, 7, 9, 10]
```

Questo è un esempio molto brutto per un sort. Infatti contiene 2 for annidati, che corrispondono ad una complessità di ordine $O(n^2)$ (che fa schifo).

Esistono diversi algoritmi di sort, di seguito alcuni esempi:

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$ variation is n	$n \log n$	n^2	$\log n$ on average, worst case space complexity is n; Sedgewick variation is $\log n$ worst case.	Typical in-place sort is not stable; stable versions exist.	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n A hybrid block merge sort is $O(1)$ mem.	Yes	Merging	Highly parallelizable (up to $O(\log n)$) using the Three Hungarians' Algorithm[7] or, more practically, Cole's parallel merge sort) for processing large amounts of data.
In-place merge sort	—	—	$n \log^2 n$ See above, for hybrid, that is $n \log n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[8]
Heapsort	n If all keys are distinct, $n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, in the worst case over sequences that have d inversions.
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space or when using linked lists.[9]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes n comparisons when the data is already sorted or reverse sorted.
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes n comparisons when the data is already sorted or reverse sorted.
Shell sort	$n \log n$	Depends on gap sequence	Depends on gap sequence; best known is $n^{4/3}$	1	No	Insertion	Small code size, no use of call stack, reasonably fast, useful where memory is at a premium such as embedded and older mainframe applications. There is a worst case $O(n(\log n)^2)$ gap sequence but it loses $O(n \log n)$ best case time.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Binary tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Cycle sort	n^2	n^2	n^2	1	No	Insertion	In-place with theoretically optimal number of writes.
Library sort	n	$n \log n$	n^2	n	Yes	Insertion	
Patience sorting	n	—	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences in $O(n \log n)$.
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection	An adaptive variant of heapsort based upon the Leonardo sequence rather than a traditional binary heap.
Strand sort	n	n^2	n^2	n	Yes	Selection	

Figure 1.11: Alcuni algoritmi di sort, tabella presa dalla pagina di Wikipedia `https://en.wikipedia.org/wiki/Sorting_algorithm`

Python come algoritmo di sort usa **Timsort**:

```
l = [10, 1, 5, 2, 7, 3, 9, 4]
print(l)
l.sort()
print(l)

[Output]
[10, 1, 5, 2, 7, 3, 9, 4]
[1, 2, 3, 4, 5, 7, 9, 10]
```

- Hybrid algorithm, derived from merge sort and insertion sort
  - Find subsequences of the data that are already ordered

  - Use that knowledge to sort the remainder more efficiently
- "Although practicality beats purity" (The Zen of Python)

References

- https://en.wikipedia.org/wiki/Algorithm
- https://discrete.gr/complexity/
- https://wiki.python.org/moin/TimeComplexity
- https://en.wikipedia.org/wiki/Sorting_algorithm
- https://bugs.python.org/file4451/timsort.txt
- https://en.wikipedia.org/wiki/Timsort

Lecture basic 7: Numpy e Scipy

- Among (many) other things, numpy offers:
  - a powerful n-dimensional array object
  - mathematical functions that interoperate natively with arrays
- And scipy provides:
  - integration
  - optimization (a.k.a. fitting)
  - interpolation
  - signal processing

Array di numpy

Numpy fornisce un'implementazione efficiente di array.

Che differenza c'è tra una lista di python e gli array di numpy? La prima differenza fondamentale è che gli array di numpy sono tendenzialmente **omogenei** (dentro lo stesso array non posso mescolare due tipi).

```
import numpy as np
# Initialization from a list
a1 = np.array([1., 2., 3])
print(a1)
# Zeros, ones, and fixed values
a2 = np.zeros(10)
a3 = np.ones((2, 2))
a4 = np.full(7, 3.)
print(a2)
print(a3)
print(a4)
# Grids
a5 = np.linspace(0., 10., 11)
a6 = np.logspace(0., 1., 11)
print(a5)
print(a6)

[Output]
[1. 2. 3.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[[1. 1.]
```

```
 [1. 1.]]
[3. 3. 3. 3. 3. 3. 3.]
[ 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.]
[ 1.         1.25892541 1.58489319 1.99526231 2.51188643 3.16227766
3.98107171 5.01187234 6.30957344 7.94328235 10.        ]
```

Altro esempio: `a = np.linspace(1, 10, 10, dtype=int)`
Se so che tipo di dati contiene un array, so già in partenza quanta memoria occupa. Questi array di numpy, una volta stanziati, tipicamente mantengono le stesse dimensioni.

numpy arrays vs. Python lists

```
import numpy as np
# arrays and lists seem similar...
l = [1., 2., 3.]
a = np.array(l)
print(l)
print(a)
# ...but they support basic arithmetic in a different fashion
print(l + l)
print(a + a)

[Output]
[1.0, 2.0, 3.0]
[1. 2. 3.]
[1.0, 2.0, 3.0, 1.0, 2.0, 3.0]
[2. 4. 6.]
```

- arrays and lists are fundamentally different objects
  - different footprint in memory, operate at different speed
  - arrays are homogeneous, lists don't need to
  - arrays offer a much more powerful indexing/slicing
  - arrays interoperate with numpy mathematical functions

Broadcasting

Broadcast vuol dire che su questi array posso fare delle operazioni (ad es somma di due array di stessa lunghezza).

```
import numpy as np
a1 = np.array([1., 2.])
a2 = np.array([[1., 2.], [3., 4.]])
c = np.pi
print(a1)
print(a2)
print(c)
print(a1 * a1)
print(a1 * c)
print(a1 * a2)
[Output]
[1. 2.]
[[1. 2.]
[3. 4.]]
3.141592653589793
[1. 4.]
[3.14159265 6.28318531]
[[1. 4.]
[3. 8.]]
```

Under certain conditions, numpy can make operations on arrays of different shape. This is extremely useful when vectorizing problems.

Con numpy posso fare cose fantasmagoriche del tipo:

```
c = np.array([[1,2],[3,4]]\\
c = np.linspace(1,16,16).reshape((4,4))
```

Nota: Ogni volta che operiamo su array, l'operazione avviene in C (perché il C è molto più veloce di python). Sostituire un loop esplicito in python con un operazione tra array di numpy è una cosa ottima in termini di efficienza! Questa cosa si chiama **vettorizzazione**.
Consideriamo il seguente ciclo for:

```
for v1, v2 in zip(v1, v2):
s += vi * v2}
```

dove `zip( , )` serve per looppare in contemporanea su due cose.
Posso fare la stessa operazione usando gli array di numpy:

```
s = (v1 * v2).sum()
```

Ho vettorizzato il problema. Cioè sono passato da un ciclo for ad un'operazione tra array. Ho ritrovato la velocità del C, mantenendo l'usabilità di python!

Lun 3 ott - Lezione 4

Lecture basic: 5 - OOP[7] introduction (1/2)

Una variabile, ad esempio un intero, funge da contenitore per un dato. Ci sono delle strutture dati, come liste e dizionari, che, oltre a contenere dei dati, sono caratterizzate da un set di operazioni che posso fare su di esse.

- Working with containers like lists or dictionaries, you may have noticed that they can do many thing besides holding the data
  - You can extend a list using append() or insert()
  - Trying to access a non-existent index in a list triggers a specific errror (*IndexError*)
  - You can iterate on a list using the handy for-loop Python syntax
  - and so on...
- In other words, a list is a variable that, in addition to its data, shows some kind of specific *behaviour*.
- How is that implemented?

Si creano delle entità di codice (ogetti) che uniscono ai dati delle funzionalità per operare su di essi. Quindi non solo definiscono come è fatto quel dato in memoria, ma anche come si manipola quel dato.
L'idea di base è tenere il codice che opera su i dati e i dati stessi in un unica entià: **l'oggetto.**

Un oggetto è un'entità di codice caratterizzata da:

- **Stato** → dati (attributi o membri)
- **Comportamento** → implementato tramite funzioni (metodi)

La programmazione a oggetti è usatissima, anche se non ovunque. Ad esempio il C non fa uso di programmazione ad oggetti.Comunque, quasi tutti i più importanti linguaggi di programmazione supportano la programmazione ad oggetti.

Classi e Oggetti

Una classe è un pezzo di codice che descrive come è fatto un oggetto. Se vogliamo programmare a oggetti dobbimo scrivere delle classi che permettono poi di creare degli oggetti.
Una classe è una generalizzazione del concetto di tipo. Ad es. il tipo "intero" si limita a dire quanto spazio occupa in memoria.
Quando scriviamo una classe dobbiamo anche descrivere le sue funzionalità (definendo delle funzioni). Una volta che abbiamo la nostra classe, ad esempio `"studente"`, possiamo creare uno o più oggetti di tipo `"studente"`.

- Basic defnitions:
  - A class is a blueprint for creating objects
  - An object is a concrete relization of a class

[7] In informatica, la programmazione orientata agli oggetti (in inglese object-oriented programming, in acronimo OOP) è un paradigma di programmazione che permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi. Particolarmente adatta nei contesti in cui si possono definire delle relazioni di interdipendenza tra i concetti da modellare (contenimento, uso, specializzazione), un ambito che più di altri riesce a sfruttare i vantaggi della programmazione ad oggetti è quello delle interfacce grafiche.

- You can imagine a class like a project, which is used to describe how objects are built and how they works
- You can have multiple objects of the same class
- The relationship is similar to the one between types and variables:
  - A type is an abstract concept, describing how a varibale is represented in memory
  - A variable is a concrete realization of it
  - You can have several variables of the same type (like several integers or several strings)
- Indeed, to some extent, a class is the generalization of the concept of type. It specifies not only how an object *is made* but also how *it behaves*.

Esempio: creiamo la classe `televisione`

- Let's consider a familiar object, like a television. It has:
  - A state
    - On/off (and possibly standby)
    - Currently displayed channel
    - Volume
    - Brightness, contrast, etc...
  - A behaviour
    - Pressing the 'power' button will turn ON/OFF
    - Rotating the volume knob will increase/decrease the volumne
    - Using the buttons on the remote control will change displayed channel, brightness, contrast etc...
    - And don't forget you need to plug-in before use!
- How would that be represented in the code?
  - The state can be represented by some attributes (variables):
    - A boolean can represent the ON/OFF state
    - For the currently displayed channel you can use an integer
    - Volume, contrast, luminosity etc...they all get their own variable(s)
  - The behaviour can be implemented through the methods:
    - For example the turn_on() and turn_off() functions may change the value of the variable and also produce all the related changes (i.e. start/stop video and audio)
    - You will probably have the netx_channel() and previous_channel() functions for zapping and so on...
    - Of course it can be much more complex than that!
- Attributes and methods are collectively called members of the class
- Each object of a specific class is an instance of that class

Python Classes

Convenzione: le classi le scrivo con l'iniziale maiuscola.

```
# Here we define the class
class Television:
    """ Television class. I will follow the convention of starting class names
    with an uppercase. """
    pass # oops we have no code yet!

"""To create instances of a class in python we use the parenthesis operator '()'.
The syntax is similar to calling a function -- which is actually what is
happening behind the scenes, as we will see later"""
my_television = Television() # my_television is an instance of the class Television

print(type(my_television)) # Check its type

your_television = Television() # And this is another instance

# Let's check that they are really two different objects
print(my_television is not your_television)
```

```
[Output]
<class '__main__.Television'>
True
```

In python tutti i tipi sono anche classi. Non esistono tipi di bassissimo livello. Persino le funzioni sono classi.

```
# Create an integer variable
this_is_an_int = 5
# Now check its type
print(type(this_is_an_int))

# Same with a string
this_is_a_string = 'Hello world!'
print(type(this_is_a_string))

# Same with a list
this_is_a_list = ['Frodo', 'Samvise', 'Meriadoc', 'Peregrino']
print(type(this_is_a_list))

# And even a function!
def this_is_a_function():
    return 0

print(type(this_is_a_function))
```

```
[Output]
<class 'int'>
<class 'str'>
<class 'list'>
<class 'function'>
```

Metodi

Per definire un metodo, basta definire una funzione dentro una classe.
Tutti i metodi di una classe ricevono automaticamente come primo argomento l'oggetto su cui li chiamiamo (`self`).

```
class Television:
    """ Class describing a televsion.
    """
    def turn_on(self, channel=1): # Class method
        """All the class methods get the object instance as their first argument.
        It is customary to call this argument 'self', though is not required
        by the language rules (you can call it 'pippo' and it will work
        just as well)
        """
        print('Turning on {}' .format(self))
        print('Showing channel {}'.format(channel))

tv = Television()
# Class methods and members are accessed through the '.' (dot) operator
# You must not pass the 'self' argument, it is added automatically!
tv.turn_on(channel=3)
```

```
[Output]
Turning on <__main__.Television object at 0x7fc718217470>
Showing channel 3
```

Attributi

```
class Television:
    """ Class describing a televsion.
    """
    pass

tv = Television()
# Add an attribute manually, with a simple assignment
# Attributes are accessed through the '.' (dot) operator
tv.x = 1
print (tv.x)
# This attribute is not shared with other instances of the class
another_tv = Television()
print(another_tv.x)
```

```
[Output]
1
Traceback (most recent call last):
File "snippets/class_attributes_1.py", line 13, in <module>
print(another_tv.x)
AttributeError: 'Television' object has no attribute 'x'
```

```
class Television:
    """ Class describing a televsion.
    """
    def add_an_attribute(self):
```

```
        """ Add a class attribute (remember the meaning of 'self') """
        self.current_channel = 1

tv = Television()
# Add an attribute from inside a class method
tv.add_an_attribute()
print (tv.current_channel)

# Again, attributes are not shared
another_tv = Television()
another_tv.add_an_attribute()
# Changing the attribute for one will not affect other instances of the class
tv.current_channel = 5
# The following line will print 1, not 5
print(another_tv.current_channel)
```

```
[Output]
1
1
```

Costruttore

- Adding attributes like that would be crazy... what would happen if I forgot to call the 'add_a_class_attribute()' method in the previous example?
- Luckily there is a solution for that: the class constructor
- The constructor is a special method that is called automatically each time a class instance is created
- A specificity of the constructor is that it cannot return anything
- In Python the constructor is the $__init__$ method[8]
- Class methods like $__init__$, with the name surronded by two underscores, are called special methods or dunder methods.
- Is is good practice to define all your class attributes inside the constructor!

```
class Television:
    """ Class describing a televsion.
    """
    def __init__(self, owner):
        """ The special method __init__ is called each time a class istance is
        created.  We can pass arguments to the constructor, just like any
        function."""
        print('Creating a television instance...')
        self.model = 'Sv32X-553T' # This class attribute is hard-coded
        self.owner = owner # This is set to the value of the argument

    def print_info(self): # Let's see
        """ Print the model and owner"""
        message = 'This is television model {}, owned by {}'
        print(message.format(self.model, self.owner))

```

[8] Actually the real constructor – that is the function responsible for creating the class instances – is the $__new__$ operator, but 99% of the time you don't need to define that, as all classes have a default one which does the job for you

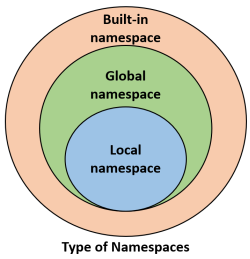
```
my_television = Television('Alberto')
my_television.print_info()
batman_television = Television('Batman')
batman_television.print_info()
```

```
[Output]
Creating a television instance...
This is television model Sv32X-553T, owned by Alberto
Creating a television instance...
This is television model Sv32X-553T, owned by Batman
```

Namespaces

Python dietro le quinte è basato su dei dizionari.

A namespace in Python is a essentially a ditcionary of *unique* names, each one associated to an object (which can be anything: a variable, a function, a class etc...).



Python creates separate namespaces for many things: for example, each time a function is called a namespace for local variables is created.
You can access objects in the local namespace (and those above – see picture) just with their name(s); for others you need the '.' (dot) operator.

The space of visibility of a variable is called it scope.

Instance attributes vs class attributes

- Each class has a namespace. Plus, each instance of the class get its own additional namespace
- The class namespace is automatically visible from each instance namespace, but not the opposite
- An attribute in an instance namesespace is an instance attribute, and cannot be seen or modified by other instances of the class
- An attribute in the class namespace is a class attribute and is shared among all the instances
- Since class attributes are not related to a specific instance, they can be accessed without creating one!
- Class attributes are useful to set class constants, or default values, or share data among instances

Class attributes (and their strange behaviour)

```
class Television:
    """ Class describing a televsion.
    """
    NUMBER_OF_CHANNELS = 999 # This is a class attribute
```

```
5
6  # We don't need an instance to access class attributes
7  print(Television.NUMBER_OF_CHANNELS)
8  # But we can also access it through instances
9  tv = Television()
10 print(tv.NUMBER_OF_CHANNELS)
11
12 # Changing the attribute in the class namespace will change it for every instance
13 another_tv = Television()
14 Television.NUMBER_OF_CHANNELS = 998
15 print(another_tv.NUMBER_OF_CHANNELS)
16 # But assigning to that attribute in an instance namespace will create a copy!
17 # Result: the other instances won't be affected!
18 tv.NUMBER_OF_CHANNELS = 997
19 print(another_tv.NUMBER_OF_CHANNELS)
```

```
1 [Output]
2 999
3 999
4 998
5 998
```

Ricapitolando:

- Object Oriented Programming (OOP) is a widespread programming paradigm, supported by many programming languages (old and modern), including Python
- An object has a state and a behaviour, represented by member variables (attributes) and member functions (methods) respectively
- A class is a blueprint for creating objects, each object is an instance of a class
- In Python classes are defined with the 'class' keyword and instanciated with the '()' operator
- Class attributes and methods (globally called members) are accessed through the '.' operator
- All the class methods get the object instance as their first argument (usually named 'self')
- You should declare class attributes in the constructor a.k.a. the $\_\_init\_\_$ function
- Instance attributes are not shared: each instance has its own copy of the data
- Class attributes are declared outside methods and are shared among all the instances of a class

Encapsulation - hidden state and interfaces

C'è un'importante differenza tra interfaccia e implementazione.
Pensiamo sempre all'esempio della televisione:

- Note that part of the state is hidden from the user (E.g. internal switches, transistors, etc. . .)
- You do not need to know what's going on inside the case to operate a TV!
- All you need to know is how to use the interface (the remote control, the knobs, the power button, the plug. . .)
- The implementation details are hidden: only the TV producer cares about them, not the user.
- This leads us to the concept of encapsulation
  - *The state of an object should only be accessed and altered through its publicly exposed interface*
- That way it is easier to find bugs: you know that, if something is wrong with an object, the problem lays inside the class code
- That way you can also *enforce behaviour*: for example you can prevent from changing the channel if the TV is off

Nota: una variabile privata è accessibile solo dall'interno della classe.

Enforcing behaviour

```
class Television:
    """ Class describing a televsion.
    """
    def __init__(self):
        """ Class constructor"""
        self.is_on = False
        self.current_channel = 1

    def turn_on(self):
        """ Turn on the tv (I omit the turn_off() method for brevity)"""
        print('Turning on!')
        self.is_on = True

    def next_channel(self):
        """ Go to next channel. Works only if the tv is on! """
        if (self.is_on):
            self.current_channel += 1

tv = Television()
tv.next_channel() # This will do nothing
print(tv.current_channel)
tv.turn_on()
tv.next_channel() # This will work
print(tv.current_channel)
```

```
[Output]
1
Turning on!
2
```

At this point you may be wondering: I can read and modify any class attribute from outside the class using the '.' (dot) operator! Doesn't that break encapsulation? Yes it does - but there are ways to fix it!

Pythonic encapsulation

- In languages like C++ you can explicitly declare that some class attributes (and methods) are *private*
- In Python there is no concept of *enforced* private attributes
- However, there exists a convention that any attribute/method name prepended by one or two underscore(s) should be considered "private"
- It's like a warning for the class user: you should never access that directly!
- In the case of two underscores Python will actually do a subtle thing to help keeping the data private – it will prepend *_classname* to the actual attribute name (see next example)
- However, not everyone in the Python community loves that
- "Never, ever use two leading underscore. This is annoyngly private"
  [*Alex Martelli*, member of the Python Software Foundation, author of 'Python in a Nutshell' and co-author of 'The Python cookbook']

"Private" attributes in Python

```
class Television:
    """ Class describing a televsion.
    """
    def __init__(self, owner):
        """ Class constructor"""
         # Single underscore - tells the user he shouldn't access the variable
         # directly outside the class
        self._model = 'Sv32X-553T'
         # Double underscore - python will prepend _Television to the name
        self.__owner = owner


tv = Television('Alberto')
 # The following line is bad practice, but it's technically possible
print(tv._model)
 # Even with two underscores I can still access it if I know the "trick"
print(tv._Television__owner)
```

```
[Output]
Sv32X-553T
Alberto
```

Pythonic encapsulation with properties

- The possibility of making variables "private" (enforced or not) is not enough of course, because sometimes we still want to let the user read or even modify the value of the attribute
- The "old" solution for that is providing access functions (the infamous "getters/setters")
- But the awsome solution is using properties (since Python 2.2)

- Properties look similar to getters and setters, but with a twist: you keep accessing the attribute with the dot operator
- In order to understand why this makes a *huge* difference let's start with an example: suppose you have a class for 2-D vectors

```
import math

class Vector2d:
    """ Class representing a Vector2d. We use float() to make sure of storing
    the coordinates in the correct format """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def module(self):
        return math.sqrt(self.x**2 + self.y**2)

    def angle(self):
        return math.atan2(self.y, self.x)

v = Vector2d(3., -1.)
print(v.x, v.y)
print(v.module(), v.angle())
```

```
[Output]
3.0 -1.0
3.1622776601683795 -0.3217505543966422
```

- Suppose you later realize that you use your Vector2d a lot for performing rotations
- It would be much faster to store the angle and the module, instead of x and y, as the rotation would reduce to a simple addition of the angles
- You may think of rewriting your class in that way...

```
import math

class Vector2d:
    """ Class representing a Vector2d - storing module and angle."""
    def __init__(self, module, angle):
        self.module = float(module)
        self.angle = float(angle)

    def x(self):
        return self.module * math.cos(self.angle)

    def y(self):
        return self.module * math.sin(self.angle)

v = Vector2d(3.1622776601683795, -0.3217505543966422)
print(v.module, v.angle)
print(v.x(), v.y())
```

In questo caso `module` e `angle` sono attributi, mentre `x` e `y` sono funzioni.

```
[Output]
3.1622776601683795 -0.3217505543966422
3.0 -1.0
```

- ... now, however, you have a big problem: your old code is broken!
- In every place where your were calling *v.x* or *v.module()* now you get an error that needs to be fixed
- If other people use your code that is even worse, because you are breaking *their* code
- Without properties the solution would have been to design the class from the start with private attributes and access functions

Old-style encapsulation: never do that!

```
import math

class Vector2d:
    """ Class representing a Vector2d. Old-style encapsulation."""
    def __init__(self, x, y):
        self._x = float(x)
        self._y = float(y)

    def x(self):
        return self._x

    def y(self):
        return self._y

    def module(self):
        return math.sqrt(self._x**2 + self._y**2)

    def angle(self):
        return math.atan2(self._y, self._x)

v = Vector2d(3., -1.)
print(v.x(), v.y())
print(v.module(), v.angle())
```

```
[Output]
3.0 -1.0
3.1622776601683795 -0.3217505543966422
```

- The class data are now "encapsulated", but that is still not ideal:
  1. You have written a lot of code just to provide access to a bunch of variables
  2. You will have to write even more methods if you want to let the user modify that variables as well (e.g. *set_x()*, *set_y()* and so on)
  3. You have to write all this code right from the start, even if you never need it, otherwise you run the risk of getting screwed later
- properties solve the problem by *emulating attributes*

Properties to emulate attributes

La property emula l'esistenza di un attributo.

```
import math

class Vector2d:
    """ Class representing a Vector2d - storing module and angle."""
    def __init__(self, module, angle):
```

```
6           self.module = float(module)
7           self.angle = float(angle)
8
9       @property
10      def x(self):
11          return self.module * math.cos(self.angle)
12
13      @property
14      def y(self):
15          return self.module * math.sin(self.angle)
16
17  v = Vector2d(3.1622776601683795, -0.3217505543966422)
18  print(v.module, v.angle)
19  # We don't need to call v.x() anymore - we can simply use v.x!
20  print(v.x, v.y)
```

```
1   [Output]
2   3.1622776601683795 -0.3217505543966422
3   3.0 -1.0
```

Setter properties

```
1   import math
2
3   class Vector2d:
4       """ Class representing a Vector2d - storing module and angle."""
5       def __init__(self, module, angle):
6           self.module = float(module)
7           self.angle = float(angle)
8
9       @property
10      def x(self):
11          return self.module * math.cos(self.angle)
12
13      @property
14      def y(self):
15          return self.module * math.sin(self.angle)
16
17      @x.setter
18      def x(self, x): # this function must be called as the property
19          """ Here we actually need to update module and angle"""
20          self.module, self.angle = math.sqrt(x**2 + self.y**2),\
21                                    math.atan2(self.y, x)
22
23  v = Vector2d(3.1622776601683795, -0.3217505543966422)
24  print(v.x)
25  v.x = 1.
26  print(v.x)
```

Alla riga 25, dietro le quinte sto chiamando il setter (x è un attributo emulato).

```
1   [Output]
2   3.0
3   1.0000000000000002
```

Make attributes read-only using properties

```
class Television:
    """ Class describing a televsion.
    """
    def __init__(self, owner):
        """ Class constructor"""
        self._owner = owner # owner is private

    @property
    def owner(self):
        return self._owner

    @owner.setter
    def owner(self, new_owner):
        """ Make the attribute read-only by acting on the setter"""
        print('Nope {}. Do you want to steal my tv?'.format(new_owner))

tv = Television('Batman')
print('This tv belongs to {}'.format(tv.owner))
tv.owner = 'Joker'
print('This tv belongs to {}'.format(tv.owner))
```

```
[Output]
This tv belongs to Batman
Nope Joker. Do you want to steal my tv?
This tv belongs to Batman
```

Final note on properties

- Bottom line: when writing a class, you don't need to make attributes private right from the start
- You can start with public attributes, and use properties later to enforce access/modification rules
- That way you only write the code that you really need

Ricapitolando: property permette di fingere che esistano attributi che in realtà non esistono.
Inizio con variabili pubbliche. Se in futuro decido di trasformarle in private, definisco una property.

Quando scriviamo una classe partiamo con tutte le variabili pubbliche. Dopodiché, se ci accorgiamo ad esempio che una variabile sia di sola lettura: la rendo privata, definisco la property di lettura e poi, volendo, definisco la property di scrittura in modo che mi restituisca un errore.

Interfaccia vs Implementazione

Quando scriviamo un codice dobbiamo pensarlo in termini dell'interfaccia.
L'interfaccia pubblica deve essere più stabile possibile: scrivo il codice in modo da non cambiare l'interfaccia. Nel caso di una classe, gli attributi pubblici fanno parte dell'interfaccia.

- A physicist thinks:
  - "I have this super-cool algorithm to solve the problem I am working on: I will code it carefully, than put together quickly some basic interface to pass data to it and write results to screen / file. I need the results quickly for my paper; I can always improve the interface later, right?"
- A programmer thinks:
  - "I will create a nice interface for the user to handle input/output in different formats and I will try to keep it as stable as possible in the future. I will start with no algorithm at all – I will just use random numbers to test the interface. I can always implement the actual algorithm later, right?"
- You don't need to think like a programmer - doing physics is your goal - but remember that interfaces are important
- The concept of interface does not just apply to the program as a whole: every significant portion of code (function, class) has its interface
- The interface of a class in Python is made by all its "public" members (methods and attributes) – i.e. those without an underscore at the beginning of their name
- Changing the interface may break every other piece of code that uses it. You want to do that *as less as possible*
- You should not access "private" members directly - even if you can. Always pass through the interface

Short summary

- Encapsulation is the technique of hiding part or all the class state to the user; he can only access and modify that through the class methods
- Encapsulation helps debugging by limiting the number of places in the code that can mutate the state of an object
- It can also be useful to enforce behaviour
- Encapsulation in Python is not enforced by the language, but rather relies on conventions
- Class members with an underscore at the beginning of their name are considered 'private' and should not be accessed directly oustide the class
- You can use properties to encapsulate your data at any moment in time - never use 'getters' and 'setters'
- Interfaces should not change frequently!

Ereditarietà

L'idea dell'ereditarietà è quella di riutilizzare il più possibile il codice. Funziona creando funzioni specializzate di una classe di partenza.
Nota: l'ereditarietà è transitiva.

- Suppose for a moment that you are coding the Monte Carlo for a physics experiment
- You want to simulate interactions of charged particles in some detector using OOP paradigm
- You may have a class Detetctor and a class for each particle that you need to simulate
- Let's say you have a class Electron, a class Positron, a class Proton and a class Alpha
- If you think about it, these classes will have a lot of code in common
- For example they all need to store their mass, charge, position, velocitiy (or momentum), possibly spin etc...
- They may also have similar behaviour, though that is less obvious
- We know that duplicate code is evil (DRY): how do we avoid that?
- Many languages - including Python offer a solution for that: inheritance
- A class can inherit from another one, automatically obtaining all its functionalities (attributes and methods) and then extending or specialyzing them
- The class which we inherit from is called *Base* class, *Parent* class or (in Python) *Superclass*
- The class inheriting is called *Derived* class or *Child* class
- In our problem we can imagine to have a base class 'Particle' and many specialized classes inheriting from it
- Inheritance is transtive: if class C inherits from class B, and class B inherits from class A, then class C is also a child of class A (and posses all its functionalities)

Inheritance: a basic example

Nel costruttore della classe figlia si chiama il costruttore della classe madre esplicitamente. Nota: questa chiamata è un po' diversa.

```
1  import math
2
3  class Particle:
4      """ Class describing a generic particle.
5      """
6      def __init__(self, mass, charge=0, name=None, momentum=0.):
7          """ Class constructor"""
8          self.mass = mass # in MeV
9          self.charge = charge # in e
10         self.name = name
11         self.momentum = momentum # in MeV/c
12
13     def energy(self):
14         """ Return the energy of the particle in MeV/c^2"""
15         return math.sqrt(self.momentum**2. + self.mass**2.)
16
17 class Electron(Particle):
18     """ Class describing an electron. We inherit from Particle
19     """
20     def __init__(self, momentum=0.):
21         """ Derived class constructor. We call the base class constructor"""
22         Particle.__init__(self, 0.511, -1., 'e-', momentum)
23
```

```
el = Electron(momentum=1.)
print('Energy of {} is {:.4f} MeV/c^2'.format(el.name, el.energy()))
```

```
[Output]
Energy of e- is 1.1230 MeV/c^2
```

Overload

Overload dei metodi: lo stesso metodo lo definiamo sia nella classe madre che nelle classi figlie. Facendo ciò, succede che il metodo nella classe figlia "sovrascrive" il metodo nella classe base.

```
class Animal:
    def sound(self):
        return None

class Dog(Animal):
    def sound(self):
        """ This will shadow the method in the base class"""
        return 'Woof!'

class Cat(Animal):
    def sound(self):
        """ This will shadow the method in the base class"""
        return 'Meow!'

class SilentAnimal(Animal):
    pass # I make no sound

animals = [Animal(), Cat(), Dog(), SilentAnimal()]
for animal in animals:
    print(animal.sound())
```

```
[Output]
None
Meow!
Woof!
None
```

Ereditarietà multipla

```
class AudioDevice:
    def play(self, channel):
        print('You are listening to channel n. {}'.format(channel))

class VideoDevice:
    def play(self, channel):
        print('You are looking to channel n. {}'.format(channel))

# Multiple inheritance!
class Television(AudioDevice, VideoDevice):
    def show(self, channel):
        AudioDevice.play(self, channel)
        VideoDevice.play(self, channel)

tv = Television()
```

```
tv.show(5)
# Is this a good idea?
tv.play(5) # Which one do we get? Why?
# Hint
# print(Television.mro())
```

mro sta per *resolution order*: la classe da cui eredito per prima è la prima in questo ordine di precedenza.

```
You are listening to channel n. 5
You are looking to channel n. 5
You are listening to channel n. 5
```

Nota: Don't abuse inheritance!
In generale: più i pezzi di codice sono indipendenti e più è facile programmare.

Composizione

Ho come attributo di una classe un oggetto di un'altra classe.

- Composition is a different technique for reusing functionalities
- The concept is simple: just use an object of some class as a member of a different one
- For example we can create the classes 'Enigne' and 'Wheel' and than the class 'Car' will have a member of type Engine and 4 members of type Wheel
- A class like 'Car' in the example is sometimes called an aggregate class

```
class Engine:
    """ Class describing a fuel engine
    """
    def start(self):
        """ Start the engine """
        print('Broom broom!')

class Car:
    """Class describing a car.
    """
    def __init__(self):
        self.engine = Engine()

    def drive(self):
        """ Start the car """
        self.engine.start()


ferrari = Car()
ferrari.drive()
```

```
[Output]
Broom broom!
```

Composition vs Inheritance

la composizione si usa quando voglio rappresentare una classe di appartenenza. Mentre con l'ereditarietà rappresento una relazione diversa: un elettrone eredita dalla classe particella perché un elettrone è una particella.

- Composition models a 'has-a' relation in the real world: a Car *has* a Engine
- Inheritance models a 'is-a' relation in the real world: an Electron *is* a Particle
- It may not always be obvious which one to use in your specific case: choose wisely!

Se siamo nel dubbio è meglio usare la composizione: è più safe.

Pitfalls of Inheritance

- Inheritance is a wild beast. There are entire libraries written about how and when (not) to use it

- Question for you: should a Square inherits from a Rectangle?
- Seems legit: a Square *is* a specialized Rectangle
- But what happens if the Rectangle class has a changeHeight() method?

- Liskov Substitution Principle: you should always be able to use a derived class instead of a base class in your code
- In other words: a derived class should always extend or specialize the functionalities of the base class, never restrict them!

Short summary

- A class can inherit funcionalities from one or more other classes (Inheritance)
- The class that inherits is call Derived (or Child) class the inherited one is the Base (or Parent) class
- Inheritance models an *is-a* relationship
- Classes can also incorporate other objects as class members (Composition)
- Composition models an *has-a* relationship
- Inheritance is tricky: use it with care!

Gio 6 ott - Lezione 5

Lecture basic: 5 - OOP Introduction (2/2)

- Suppose we want to create a class for managing 2D vectors[9]
- That's just for learning: there are already plenty of libraries for doing array operations - like numpy!
- Anyway let's start coding some useful methods for it

```
import math

class Vector2d:
    """ Class representing a Vector2d. We use float() to make sure of storing
    the coordinates in the correct format """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def module(self):
        return math.sqrt(self.x**2 + self.y**2)

    def info(self):
        print ('Vector2d({}, {})'.format(self.x, self.y))

    def add(self, other):
        return Vector2d(self.x + other.x, self.y + other.y)

v = Vector2d(3., -1.)
v.info()
print(v.module())
z = Vector2d(1., 1.5)
t = v.add(z)
t.info()
```

```
[Output]
Vector2d(3.0, -1.0)
3.1622776601683795
Vector2d(4.0, 0.5)
```

- This kind of works but..... isn't that ugly?
- Look at the lines *v.info()* or *v.module()*. It would be far more readible to just do *print(v)* and *abs(v)*
- And what about $t = v.add(z)$? Why not $t = v + z$?
- In Python there is a tool that allows you to do just that: special methods
- Last lesson we saw that special methods (or dunder methods or magic methods) are methods like *__init__* and got a special treatment by the Python interpeter
- There are a few tens of special methods in Python. Let's see how they work

[9] The content of this lesson is vastly based on the the book '*Fluent Python*' by Luciano Ramalho

Special Methods

```
import math

class Vector2d:
    """ Class representing a Vector2d """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __abs__(self):
        # Special method!
        return math.sqrt(self.x**2 + self.y**2)

v = Vector2d(3., -1.)
# The Python interpeter automatically replace abs(v) with Vector2d.__abs__(v)
print(abs(v))
```

```
[Output]
3.1622776601683795
```

- And what about *print()*?
- There are actually two special methods used for that: *__str__* and *__repr__*
- *__str__* is meant to return a concise string for the user; it is called with *str()*
- *__repr__* is meant to return a richer output for debug. It is called with *repr()*
- *print()* automatically tries to get a string out of the object using *__str__*
- If there isn't one, it searches for *__repr__*. A defealut *__repr__* is automatically generated for you, if you haven't defined one

__str__ and *__repr__*

```
class Vector2d:
    """ Class representing a Vector2d """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __repr__(self):
        # We don't want to hard-code the class name, so we dynamically get it
        class_name = type(self).__name__
        return ('{}({}, {})'.format(class_name, self.x, self.y))

    def __str__(self):
        """ We convert the coordinates to a tuple so that we can reuse the
        __str__ method of tuples, which already provides a nice formatting.
        Notice the two parenthesis: this line is equivalent to:
        temp_tuple = (self.x, self.y)
        return str(temp_tuple)
        """
        return str((self.x, self.y))

v = Vector2d(3., -1.)
print(v) # Is the same as print(str(v))
print(repr(v))
print('I got {} with __str__ and {!r} with __repr__'.format(v, v))
```

```
[Output]
(3.0, -1.0)
Vector2d(3.0, -1.0)
I got (3.0, -1.0) with __str__ and Vector2d(3.0, -1.0) with __repr__
```

Mathematical operations

```
class Vector2d:
    """ Class representing a Vector2d """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __add__(self, other):
        return Vector2d(self.x + other.x, self.y + other.y)

    def __mul__(self, scalar):
        return Vector2d(scalar * self.x, scalar * self.y)

    def __rmul__(self, scalar):
        # Right multiplication - because a * Vector is different from Vector * a
        return self * scalar # We just call __mul__, no code duplication!

    def __str__(self):
        # We keep this to show the results nicely
        return str((self.x, self.y))

v, z = Vector2d(3., -1.), Vector2d(-5., 1.)
print(v+z)
print(3 * v)
print(z * 5)
```

```
[Output]
(-2.0, 0.0)
(9.0, -3.0)
(-25.0, 5.0)
```

In-place operations

```
class Vector2d:
    """ Class representing a Vector2d """
    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

    def __imul__(self, other):
        self.x *= other.x
        self.y *= other.y
        return self

```

```
17      def __str__(self):
18          return str((self.x, self.y))
19
20  v = Vector2d(3., -1.)
21  z = Vector2d(-5., 1.)
22  v += z
23  print(v)
24  v *= z
25  print(v)
```

```
1   [Output]
2   (-2.0, 0.0)
3   (10.0, 0.0)
```

L'addizione fatta con `__iadd__` è concettualmente diversa da quella fatta con `__add__`. Infatti in questo caso non sto creando un nuovo vettore, ma sto modificando gli attributi del vettore su cui è chiamata la funzione.

Comparisons

In-place operations

```
1   import math
2
3   class Vector2d:
4       """ Class representing a Vector2d """
5       def __init__(self, x, y):
6           self.x = float(x)
7           self.y = float(y)
8
9       def __abs__(self):
10          # We need this for __eq__
11          return math.sqrt(self.x**2 + self.y**2)
12
13      def __eq__(self, other):
14          # Implement the '==' operator
15          return ((self.x, self.y) == (other.x, other.y))
16
17      def __ge__(self, other):
18          # Implement the '>=' operator
19          return abs(self) >= abs(other)
20
21      def __lt__(self, other):
22          # Implement the '<' operator
23          return abs(self) < abs(other)
24
25      def __repr__(self):
26          # We define __repr__ for showing the results nicely
27          class_name = type(self).__name__
28          return ('{}({}, {})'.format(class_name, self.x, self.y))
```

Alla riga 15 sfruttiamo l'uguaglianza delle *tuple* in modo da risparmiarci l'implementazione from scratch dell'uguaglianza tra float.

```
1   from vector2d_comparable import Vector2d
2
3   v, z = Vector2d(3., -1.), Vector2d(3., 1.)
4   print(v >= z, v == z, v < z)
5   # This works even if we don't define the __gt__ method explicitly
```

```
print(v > z)

vector_list = [Vector2d(3., -1.), Vector2d(-5., 1.), Vector2d(3., 0.)]
print(vector_list)
# Tho make the following line work we need to implement either __ge__ and __lt__
# or __gt__ and __le__ (we need a complementary pair of operator)
vector_list.sort()
print(vector_list)
# Note: we got the full power of timsort for free! Nice :)
```

```
[Output]
True False False
False
[Vector2d(3.0, -1.0), Vector2d(-5.0, 1.0), Vector2d(3.0, 0.0)]
[Vector2d(3.0, 0.0), Vector2d(3.0, -1.0), Vector2d(-5.0, 1.0)]
```

An hashable Vector2d

- Ok now let's try to make our vector2d *hashable*
- Hashable objects can be put in *sets* and used as keys for dictionaries
- To make an object hashable we need to fullfill 3 requirements:
  - It has to be immutable - otherwise you may not retrieve the correct hash
  - It needs to implement a $__eq__$ function, so one can compare objects of this class
  - It needs a (reasonable) $__hash__$ function
- Rules for a good hash function:
  - Must return the same value for objects that compare as equal
  - Should rarely return the same value for different objects
  - Should sample the result space uniformly

```
class Vector2d:
    """ Class representing a Vector2d """
    def __init__(self, x, y):
        """" We tell the user that x and y are private"""
        self._x = float(x)
        self._y = float(y)

    @property
    def x(self):
        """ Provides read only access to x - since there is no setter"""
        return self._x

    @property
    def y(self):
        """ Provides read only access to y - since there is no setter"""
        return self._y

    def __eq__(self, other):
        return ((self.x, self.y) == (other.x, other.y))

    def __hash__(self):
        """ As hash value we provide the logical XOR of the hash of the two
        coordinates """
        return hash(self.x) ^ hash(self.y)

```

```
    def __repr__(self):
        # Again we neeed __repr__ to display the results nicely
        class_name = type(self).__name__
        return ('{}({}, {})'.format(class_name, self.x, self.y))
```

```
from vector2d_hashable import Vector2d

v, t, z = Vector2d(3., -1.), Vector2d(-5., 1.), Vector2d(3., -1.)
# Check the equality
print(v == t, v == z, t == z)
# Check the hash: v and z are equal, so they will have the same hash
print(hash(v), hash(t), hash(z))
# v and t have different hash, so they can be in the same set
print({v, t})
# v and z have the same hash -- only one will be stored in the set!
print({v, z})
```

```
[Output]
False True False
-3 -6 -3
Vector2d(-5.0, 1.0), Vector2d(3.0, -1.0)
Vector2d(3.0, -1.0)
```

Array N-dimensionali

- 2d array are boring... why not a N-d array?
- Of course we cannot store the components explicitly like before
- We need a container for that and we will use *array* from the array library
- This is an example of composition
- Question for you: why not a list or a tuple? → Perché le liste sono lente, non sono pensate per fare operazioni matematiche (i dati in memoria non sono contigui)
- Note: *array* uses a typecode (a single character) for picking the type. 'd' is the typecode for float numbers in double precision.

```
import math
from array import array

class Vector:
    """ Classs representing a multidimensional vector"""
    typecode = 'd' #We use a class attribute to save the code required for array

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __repr__(self):
        """ Calling str() of an array produces a string like
        array('d', [1., 2., 3., ...]). We remove everything outside the
        square parenthesis and add our class name at the beginning."""
        components = str(self._components)
        components =  components[components.find('['): -1]
        class_name = type(self).__name__
        return '{}({})'.format(class_name, components)

    def __str__(self):
```

```
21          return str(tuple(self._components)) # Using str() of tuples as before
22
23  v = Vector([5., 3., -1, 8.])
24  print(v)
25  print(repr(v))
```

A riga 6 stiamo salvando il `typecode` come attributo della classe: ovvero è condiviso con tutte le istanze della classe.

```
1  (5.0, 3.0, -1.0, 8.0)
2  Vector([5.0, 3.0, -1.0, 8.0])
```

- Now that we have an arbitrary number of components, we cannot access them like vector.x, vector.y, ... anymore
- What we want is a syntax similar to that of lists: vector[0], vector[1] ans so on
- There are two magic methods for that: *__getitem__* for access and *__setitem__* for modifying
- While we are at it, we also implement the *__len__* method, which allows us to call *len(vector)*

```
1  import math
2  from array import array
3
4  class Vector:
5      """ Classs representing a multidimensional vector"""
6      typecode = 'd'
7
8      def __init__(self, components):
9          self._components = array(self.typecode, components)
10
11     def __getitem__(self, index):
12         """ That's super easy, as we get to reuse the __getitem__ of array!"""
13         return self._components[index]
14
15     def __setitem__(self, index, new_value):
16         """ Same as __getitem__, we just delegate to the __setitem__ of array"""
17         self._components[index] = new_value
18
19     def __len__(self):
20         """ Did I just write that we like to delegate? """
21         return len(self._components)
22
23     def __repr__(self):
24         components = str(self._components)
25         components =  components[components.find('['): -1]
26         class_name = type(self).__name__
27         return '{}({})'.format(class_name, components)
```

```
1  from vector_random_access import Vector
2
3  v = Vector([5., 3., -1, 8.])
4
5  print(len(v))
6
7  print(v[0], v[1])
8
```

```
v[1] = 10.
print(v)

print (v[9]) # This will generate an error!
```

```
[Output]
4
5.0 3.0
Vector([5.0, 10.0, -1.0, 8.0])
Traceback (most recent call last):
    File "snippets/test_vector_random_access.py", line 12, in <module>
        print (v[9]) # This will generate an error!
    File "/data/work/teaching/cmepda/slides/latex/snippets/vector_random_access.py", line 13,
        return self._components[index]
IndexError: array index out of range
```

An Iterable Vector

- Now our vector behaves a bit like a native python list
- However a list has a very powerful feature we miss: it's iterable
- An *iterable* in Python is something that has a $__iter__$ method, which returns an iterator
- Technically, an iterator is an object that implements the $__next__$ special method, which is used to retrieve elements one at a time
- We will not dicuss iterators any further here: instead, we will just exploit composition and borrow the $__iter__$ method from the underlying array

```
import math
from array import array

class Vector:
    """ Classs representing a multidimensional vector"""
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __iter__(self):
        """ We don't need to code anything... an array is already iterable!"""
        return iter(self._components)

if __name__ == '__main__':
  v = Vector([5.1, 3.7, -25.])
  for component in v:
      print(component)
```

```
[Output]
5.1
3.7
-25.0
```

Duck Typing[10]

```python
class Duck:
    """ This is a duck - it quacks"""

    def quack(self):
        print('Quack!')

class Goose:
    """ This is a goose - it quacks too"""

    def quack(self):
        print('Quack!')

class Penguin:
    """ This is a penguin -- He doesn't quack!"""
    pass

birds = [Duck(), Goose(), Penguin()]

for bird in birds:
    bird.quack()
```

```
[Output]
Quack!
Quack!
Traceback (most recent call last):
    File "snippets/duck_typing.py", line 20, in <module>
        bird.quack()
AttributeError: 'Penguin' object has no attribute 'quack'
```

Polymorphism

- Reuse the same code for different things
- In statically typed languages this is tipically done with inheritance, e.g. we make Duck and Goose inherits from a base class QuackingBird() or something like that
- Python is dynamical, so we can use duck typing for that. We just need to implment the quack() method for both Ducks() and Goose() and we are done
- In other words we obtain polymorphism just by satsisfying the required interface (in this case the quack() function)

The power of iterables

- Having an iterable Vector (thanks to the $__iter__$ magic method) makes all the difference in the world
- There are *a lot* of builtin and library functions in python accepting a generic iterable as input:
  - *sum*: Sum all the elements
  - *max*/*min*: Return the maximum/minimum
  - *enumerate*: Iterate with automatic counting of iterations
  - *map*: Apply a function to the elements one by one
  - *filter*: Iterate only on the elements passing a given condition

[10] "If it looks like a duck and quacks like a duck, it must be a duck."

  - *zip*: Iterate over pairs of elements (requires two iterables)

- Countless others can be found in the *itertools* library
  - *islice*: Slice the loop with start, stop and step
  - *takewhile*: Stop looping when a condition becomes false
  - *chain*: Loop through many sequences one after another
  - *cycle*: Loop over the sequence repeatedly, indefinitely
  - *permutations*: Get all the permutations of a given length
  - And so on...

- With duck typing we can now use any of that for our Vector class – isn't that cool?

```
from vector_iterable import Vector
from itertools import permutations

vec = Vector([1., 2., 4.])

# Select only the elements passing a given condition
def filter_function(x):
    return x > 3.

filtered = [x for x in filter(filter_function, vec)] # list comprehension
print(filtered)

# Print all the permutations of two elements
for p in permutations(vec, 2):
    print(p)
```

```
[Output]
[4.0]
(1.0, 2.0)
(1.0, 4.0)
(2.0, 1.0)
(2.0, 4.0)
(4.0, 1.0)
(4.0, 2.0)
```

A vector that behaves like a duck

```
import math
from array import array

class Vector:
    """ Classs representing a multidimensional vector"""
    typecode = 'd'

    def __init__(self, components):
        self._components = array(self.typecode, components)

    def __len__(self):
        return len(self._components)

    def __iter__(self):
        return iter(self._components)

    def __str__(self):
```

```
        return str(tuple(self)) # tuple() accept an iterable

    def __abs__(self):
        return math.hypot(*self._components)

    def __add__(self, other):
        """ zip returns a sequence of pairs from two iterables"""
        return Vector([x + y for x, y in zip(self, other)])

    def __eq__(self, other):
        return (len(self) == len(other)) and \
               (all(a == b for a, b in zip(self, other)))   # Efficient test!
```

```
from vector_ducked import Vector

v = Vector([1., 2., 3.])
t = Vector([1., 2., 3., 4.])
z = Vector([1., 2., 5.])
u = Vector([1., 2., 3.])

print(v)
print(abs(v))
print(sum(v))
print(v == t, v == z, v == u)
print(v+z)
print(v+t) # Note the result: this is due to the behaviour of zip()!
```

```
[Output]
(1.0, 2.0, 3.0)
3.741657386773941
6.0
False False True
(2.0, 4.0, 8.0)
(2.0, 4.0, 6.0)
```

Function are classes

- Remember that in the past lesson I told you that functions are objects of the 'function' class.
- How are they implemented?
- With a special method - of course: $__call__$
- Every object implementing a $__call__$ method is called callable

Esempio: Noi tipicamente a `curve_fit` passiamo una funzione, ma in realtà possiamo passarle un qualsiasi *callable*!

A simple callable for a straight line

```
class Line:
    """Class representing a straight line"""
    def __init__(self, slope=1., intercept=0.):
        self.slope = slope
        self.intercept = intercept

    def __call__(self, x):
        return self.slope * x + self.intercept
```

```
9
10      def __str__(self):
11          return 'y = {} x + {}'.format(self.slope, self.intercept)
12
13      def __repr__(self):
14          return 'Slope = {}, Intercept = {}'.format(self.slope, self.intercept)
15
16  line = Line(slope=-2., intercept=1.)
17  print(line)
18  print(repr(line))
19  print(line(2.))
```

```
1  [Output]
2  y = -2.0 x + 1.0
3  Slope = -2.0, Intercept = 1.0
4  -3.0
```

Create a call counter

```
1  class CallCounter:
2
3      """Wrap a generic function and count the number of times it is called"""
4
5      def __init__(self, func):
6          # We accept as input a function and store it (privately)
7          self._func = func
8          self.num_calls = 0
9
10     def __call__(self, *args, **kwargs):
11         """ This is the method doing the trick. We use *args and **kwargs to
12         pass all possible arguments to the function that we are wrapping"""
13         # We increment the counter
14         self.num_calls += 1
15         # And here we just return whatever the wrapped function returns
16         return self._func(*args, **kwargs)
17
18     def reset(self):
19         self.num_calls = 0
```

Il *"wrapper"* aggiunge un layer di funzionalità intermedie.
Riga 16: siccome non conosciamo quali argomenti prende questa funzione, dobbiamo fare in modo che prenda un qualsiasi numero di argomenti.

Fit hacking

```
1  import numpy
2  from scipy.optimize import curve_fit
3  import matplotlib.pyplot as plt
4  from callable import CallCounter
5
6  def line(x, m, q):
7      return m * x + q
8
9  # Generate the datasets: a straight line + gaussian fluctuations
10 x = numpy.linspace(0., 1., 20)
11 y = line(x, 2., 10.) + numpy.random.normal(0, 0.1, len(x))
```

```
12
13  # Fit
14  counting_func = CallCounter(line)
15  popt, pcov = curve_fit(counting_func, x, y, p0=[-1., -100.]) # p0 is mandatory here
16  print('Fitted with {} function calls'.format(counting_func.num_calls))
17
18  # Show the results
19  m, q = popt
20  plt.figure('fit with custom callable')
21  plt.plot(x, y, 'bo')
22  plt.plot(x, line(x, m, q), 'r-')
23  plt.show()
```

Riga 15: sto passando a `curve_fit` la funzione *"wrappata"*. In questo caso devo per forza passargli p0, altrimenti `curve_fit` non ha modo di capire il numero di parametri: tipicamente per capirlo guarda gli argomenti della funzione che gli passiamo, ma in questo caso gli stiamo passando un callable generico!

```
1  [Output]
2  Fitted with 9 function calls
```

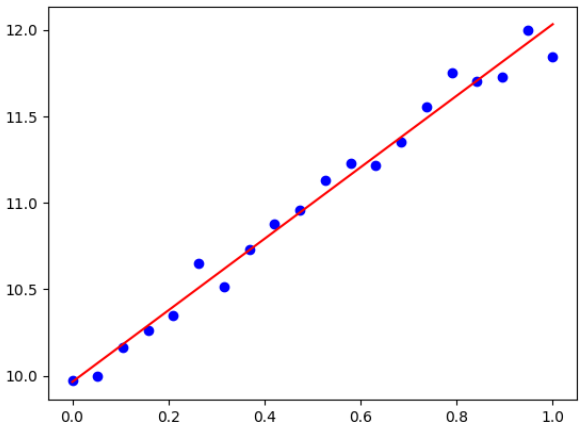


Figure 1.12: The fit works as usual

Summary:

- Special methods can beused to greatly enhance the readibility of the code
- There are tens of special methods in python, covering logical operations, mathematical operations, array-style access, iterations, formatting and many other things...
- Implementing the required interface in your classes you will be able to reuse a lot of code written for the standard containers thanks to duck typing, which is the pythonic way to polymorphism

Lun 10 ott - Lezione 6

Lecture Advanced 1: Testing and documentation

How do I make sure my program is correct?

In un linguaggio compilato alcune le verifiche le fa il compilatore, che fa per lo meno alcune verifiche. Nei linguaggi interpretati, come python è più difficile, perché fino a quando non **eseguiamo** il codice, nessuno sa che argomenti passiamo a una funzione. Si fanno due cose nei linguaggi interpretati: unit testing e static analysis (es. pylint).

- The short answer is: in real life you don't!
  - Especially if your code is asynchronous
- That is not the same a saying there is nothing you can do
- For compiled langauges the compiler will flag all obvious (and a whola lotta of non-obvious) mistakes
  - This doesn't really apply to Python, since Python is interpreted
  - Although the interpreter will stop upon syntax errors
- Besides paying attention, there are two things that you can do even in interpreted languages:
  1. Unit testing
  2. Static analysis
- Generally people hate both, but they should come right next to version control in your work-flow toolbox

Unit testing naïve example

```
1  def square(x):
2      """Function returning the suare of x.
3      """
4      return x**2.
5
6  def test():
7      """Dumb unit test---make sure that the square of 2. is 4.
8      """
9      assert square(2.) == 4.
10     print('Passed---cool!')
11
12
13 if __name__ == '__main__':
14     test()
```

```
1 [Output]
2 Passed---cool!
```

Il test in questo caso si assicura che il quadrato di 2 faccia 4. Chiaramente questo è un caso particolare. Unit testing significa spezzare il codice in unità elementari, individuare alcuni casi interessanti e verificare che in questi casi tutto funzioni a dovere.
Se facciamo crescere il nostro programma organicamente con una serie di unit test, riusciamo ad evitare molti errori!

Unit testing in a nutshell

- Break up your program in many small pieces
  - Each piece should encapsulate a well-defined and (possibly) simple functionality. *Le funzioni che fanno 100 cose insieme non vanno bene, è meglio scomporle in 100 funzioni!*
- This is usually accomplished by means of a sensible hierarchy of functions and classes
  - And this is typically the hardest task when structuring your code
  - And the code will evolve with time, so you will find yourself refactoring code from time to time
  - Remember to be dry: don't repeat youself
- Unit testing is: make sure that each single piece is correct by implementing a series of basic checks
  - You know what each elementary piece of code is suppose to be
  - Make sure it does
  - And make sure it does with any valid input
- This is much simpler that testing the whole program at once
  - Although you have to do that, too
- Test-Driven Development (TDD)
  1. Write an empty placeholder for your new function
  2. Write all the unit tests (they will fail)
  3. Implement your function and tweak it until all the tests pass

TDD: Quando scriviamo un codice, tipicamente abbiamo delle specifiche da rispettare. Scrivo prima il test in base alle specifiche; poi scrivo il corpo vuoto della funzione; e infine implemento la funzione finché passi tutti i test.

É importante scrivere test e documentazione di pari passo con la stesura del codice! Non devo aspettare la fine per farli!

Back to our naïve example

Cosa può andare storto?
Cosa succede se passo una stringa alla nostra funzione? Avrò un TypeError. In questo caso è facile individuare l'errore, ma a volte non è così semplice. É molto utile scrivere un unit test che controlli che in input alla funzione sto dando un numero.
Ci sono infiniti modi in cui una cosa potrebbe andare storto!

```
1  def square(x):
2      """Function returning the suare of x.
3      """
4      return x**2.
5
6  def test():
7      """Dumb unit test---make sure that the square of 2. is 4.
8      """
9      assert square(2.) == 4.
10     print('Passed---cool!')
11
12
13 if __name__ == '__main__':
14     test()
```

```
[Output]
Passed---cool!
```

- This is fine, but everything happens manually
  - You have to run the script yourself
  - You have to inspect the output yourself
- As your code grows in complexity, this is not very effective

[Le variabili d'ambiante sono la chiave per il funzionamento del sistema operativo.]

Unit tests the Python way: The unittest module

C'è un altra cosa che si chiama pytest, che ultimamente ha soppiantato unittest, ma la logica è la stessa.
Idealmente ogni volta che faccio una modifica vorrei runnare tutti i test.
C'è una cosa che si chiama **continous integration**.

```python
import unittest

def square(x):
    """Function returning the suare of x.

    In real life this would be in a differnt module!
    """
    return x**2.


class TestSquare(unittest.TestCase):

    def test(self):
        """Dumb unit test---make sure that the square of 2. is 4.
        """
        self.assertAlmostEqual(square(2.), 4.)


if __name__ == '__main__':
    unittest.main()
```

```
[Output]
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Wait a moment... How is this different?

- This is much better!
- The base TestCase class offers all the goodies for unit testing
  - assertTrue(), assertFalse(), assertEqual(), assertAlmostEqual()...
- The execution can be easily made automatic:
  - Put all your unit test modules into a test folder

  - Run `python -m unittest discover`
  - (Or, even better, write a small Makefile or .bat script to do that)
  - That's it—all your tests are run in sequence

- Did you just find a bug in your code?
  - Make sure you add a unit test along with the fix, so that you'll never be hurt again by that particular bug
- Are you adding a new feature?
  - Make sure the new code is covered by unit tests
  - You should not be obsessed by the coverage, but you should definitely aim for it to be as large as possible
- You should always make sure that all the unit tests are passing before merging stuff on the master
- More about this in a bit (we'll be talking about continuous integration)

Static code analysis

- By its very nature, Python will show you all the errors at runtime
- Say you have a bug in a part of the code that is exercised very rarely, and not covered by unit tests
  - Python might crash the first time you exercise it...
  - or Python might happily do *something* that is not what you intended
- It might take years for even realizing that there is a bug
- Many common mistakes can be found by just looking at the code
  - And in fact all of them can, at least in principle
- Part of it can be done programmatically
  - Generally, a program will not *understand* your program
  - But a program can be trained to spot some kind of errors and inconsistencies
- Pylint and pyflakes are good examples of such tools

Static analysis: an example

```
1  x = 1.
2  y = 2.
3  very_uncommon_condition = False
4  if very_uncommon_condition:
5      print(x + z)
6  else:
7      print(x + y)
```

```
1  [Output]
2  3.0
```

- And here is the pylint output

```
[lbaldini@nbbaldini latex]$ pylint snippets/linting1.py
************* Module snippets.linting1
snippets/linting1.py:1:0: C0111: Missing module docstring (missing-docstring)
snippets/linting1.py:1:0: C0103: Constant name "x" doesn't conform to UPPER_CASE
                          naming style (invalid-name)
snippets/linting1.py:2:0: C0103: Constant name "y" doesn't conform to UPPER_CASE
                          naming style (invalid-name)
snippets/linting1.py:3:0: C0103: Constant name "very_uncommon_condition" doesn't
                          conform to UPPER_CASE naming style (invalid-name)
snippets/linting1.py:5:14: E0602: Undefined variable 'z' (undefined-variable)

--------------------------------------------------------------------
Your code has been rated at -5.00/10 (previous run: -5.00/10, +0.00)
```

Static code analysis

- You should consider using static code analysis routinely
- Static analysis tools tend to be quite verbose
  - And often times verbose is the same as annoying
- They try and enforce many different (good!) things at once
  - Formal correctness
  - Efficiency
  - Avoiding anti-patterns
  - Style guides
  - Generic conventions
- They also are typically highly customizable
  - i.e., you can mute errors you don't care about
  - But be advised: you most of the times you should probably care
- Finding a good balance is generally not too hard
- And trust me: it will help you in the long run

Digression: optional static typing in Python

```
1  def square(x):
2      """Return the square of a number.
3      """
4      return x**2.
5
6  def annotated_square(x: float) -> float:
7      """Return the square of a number.
8      """
9      return x**2.
10
11 print(square(2.))
12 print(annotated_square(2.))
```

```
1  [Output]
2  4.0
3  4.0
```

- Recent Python 3 versions support type annotations
- The Python interpreter recognizes but does nothing with annotations
  - And so what?

- Well. . . they are handy (as comments are)
  - The code is easier to read
  - Even more checks wrt un-annotated code can be done by tools such as mypy

Continuous integration

- Imagine for a second. . .
  - Wouldn't it be nice if sombody run all the unit tests of my package every time I push on the master or make a pull request?
  - And, since we are at it, sent me an email if any of the tests fail?
- . . . Well, such a thing exists and it is standard practice in code dvelopment
- People even made a name for it: Continuous Integration (CI)
- CI cloud-base services exists just like code-hostng services exist
  - Travis-CI and circleci are two good examples
- They interoperates seamlessly with github, gitlab or bitbucket
- Setting up CI for your package is usually fairly simple
- One-sentence summary: go ahead and do it. Always.

Documentation

La documentazione vive insieme al codice! E il meccanismo che si usa per implementare il codice sono le "docstring". C'è una sintassi ben precisa che permette ad un tool automatico, che nel caso di python si chiama Sphynx, di generare la documentazione.

Ci sono vari programmi di host per la documentazione (ad esempio uno open source è **readthedocs**).

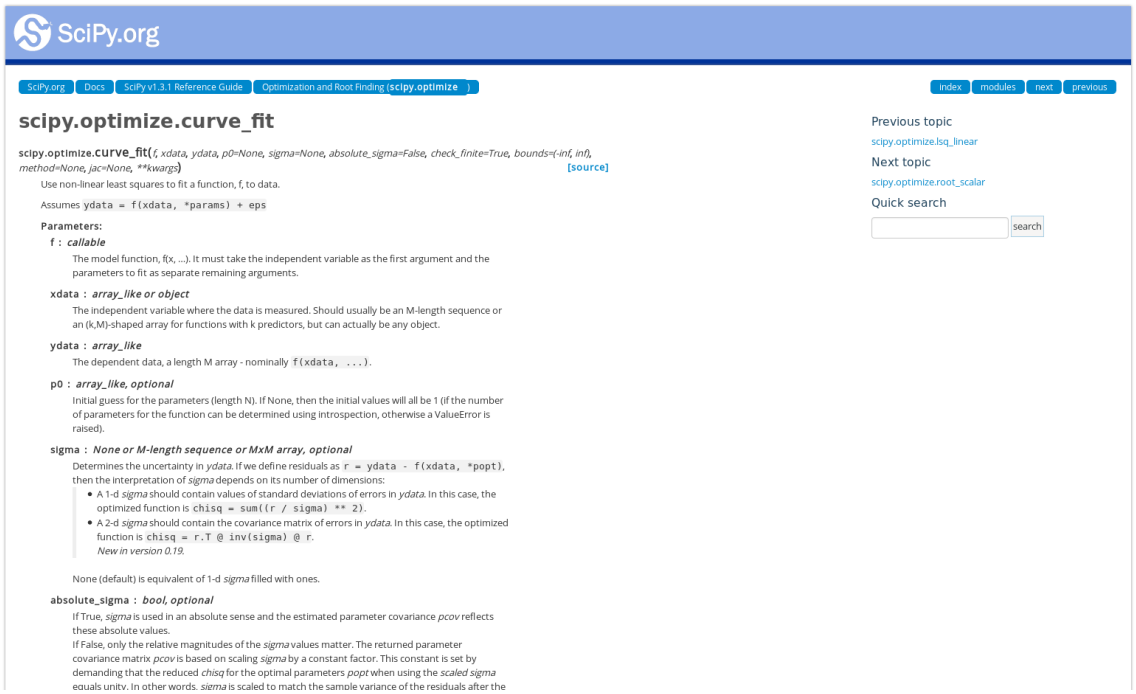


Figure 1.13: How do the hell they do that?

```
def curve_fit(f, xdata, ydata, p0=None, sigma=None, absolute_sigma=False,
              check_finite=True, bounds=(-np.inf, np.inf), method=None,
              jac=None, **kwargs):
    """
    Use non-linear least squares to fit a function, f, to data.

    Assumes ``ydata = f(xdata, *params) + eps``

    Parameters
    ----------
    f : callable
        The model function, f(x, ...).  It must take the independent
        variable as the first argument and the parameters to fit as
        separate remaining arguments.
    xdata : array_like or object
        The independent variable where the data is measured.
        Should usually be an M-length sequence or an (k,M)-shaped array for
        functions with k predictors, but can actually be any object.
    ydata : array_like
        The dependent data, a length M array - nominally ``f(xdata, ...)``.
    p0 : array_like, optional
        Initial guess for the parameters (length N).  If None, then the
        initial values will all be 1 (if the number of parameters for the
        function can be determined using introspection, otherwise a
        ValueError is raised).
    sigma : None or M-length sequence or MxM array, optional
        Determines the uncertainty in `ydata`. If we define residuals as
        ``r = ydata - f(xdata, *popt)``, then the interpretation of `sigma`
        depends on its number of dimensions:

            - A 1-d `sigma` should contain values of standard deviations of
              errors in `ydata`. In this case, the optimized function is
              ``chisq = sum((r / sigma) ** 2)``.

            - A 2-d `sigma` should contain the covariance matrix of
              errors in `ydata`. In this case, the optimized function is
              ``chisq = r.T @ inv(sigma) @ r``.

              .. versionadded:: 0.19

        None (default) is equivalent of 1-d `sigma` filled with ones.
    absolute_sigma : bool, optional
        If True, `sigma` is used in an absolute sense and the estimated parameter
        covariance `pcov` reflects these absolute values.

        If False, only the relative magnitudes of the `sigma` values matter.
        The returned parameter covariance matrix `pcov` is based on scaling
        `sigma` by a constant factor. This constant is set by demanding that the
        reduced `chisq` for the optimal parameters `popt` when using the
```

Figure 1.14: the documentation is embedded in the code. . .

Sphinx: the documentation tool for Python

Sphynx basics

- Process all the relevant information to produce several types of output
  - Most notably html and LaTeX
- Two different sources:
  1. The doctrings in the Python modules
  2. Additional markup files (in reStructuredText) containing auxiliary information
- Typical workflow:
  - Use `sphinx-quickstart` once when you setup your project
  - Tweak the generated `conf.py` file to suit your needs
  - Go ahead and have fun!
- Sphinx is *very* powerful
  - e.g., `https://docs.python-guide.org/` is written in Sphinx, and so is all the Python documentation

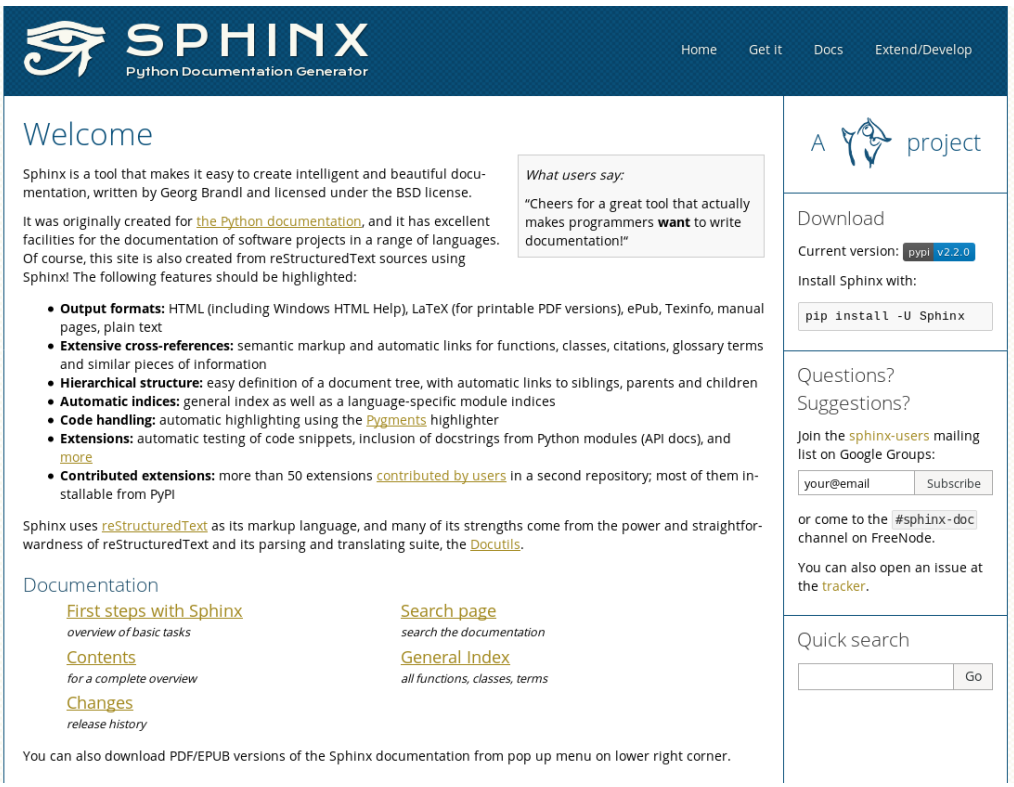


Figure 1.15: .

Ok, I have the documentation compiled, now what do I do with it?

- Wouldn't it be nice if the documentation was automatically compiled and uploaded on the web each time I push on the master?
- This is possible and is called readthedocs.com
  - And, again, this is a cloud-based service that can interoperate easily with github, gitlab or bitbucket

NOTA: Per il progetto di fine anno bisogna fare tutto questo, compreso avere la documentazione su un sito come readthedocs.

Torniamo a numpy

L'ultima volta abbiamo visto il broadcasting.

Mathematical functions in Numpy

```
import numpy as np
import math

a = np.array([0.1, 1., 10.])

print(np.log10(a))
print(np.exp(a))
print(np.sin(a))

print(np.log10(0.1))

print(math.log10(a))
```

```
[Output]
[-1. 0. 1.]
[1.10517092e+00 2.71828183e+00 2.20264658e+04]
[ 0.09983342 0.84147098 -0.54402111]
-1.0
Traceback (most recent call last):
    File "snippets/numpy_functions.py", line 12, in <module>
        print(math.log10(a))
TypeError: only size-1 arrays can be converted to Python scalars
```

numpy mathematical functions interoperate natively with arrays (and work on plain old numbers, too).

Array and Masks

Masks are a powerful tool in numpy. They can replace conditional expressions in a for loop in vectorization context .

```
import numpy as np

a = np.linspace(0., 10., 11)
mask1 = a >= 2.5
mask2 = a < 8.5

print(a)
print(mask1)
print(mask2)
print(a[mask1])
print(a[mask2])
print(a[np.logical_and(mask1, mask2)])
```

```
[Output]
[ 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.]
[False False False True True True True True True True]
[ True True True True True True True True True False False]
[ 3. 4. 5. 6. 7. 8. 9. 10.]
```

```
[0. 1. 2. 3. 4. 5. 6. 7. 8.]
[3. 4. 5. 6. 7. 8.]
```

Altro esempio:

```
a = np.random.uniform(size=10)
mask = a > 0.5

mask.sum() #mi restituisce il numero di elementi che soddisano la condizione

#posso passare una maschera tra parentesi quadre per indirizzare gli elementi di un array.
#Mi restituisce un nuovo array contenente solo gli elementi che soddisfano la condizione

a[mask]
```

Da fare: guardare come si fa lo slicing di un array

Digression: pseudo-random number generators

```
import random
x = random.random()
```

- Every programming language comes with a Pseudo Random Number Generator (PRNG)
  - Python is no exception: `https://docs.python.org/3/library/random.html`
  - Mersenne-Twister, 53-bit precision, period of $2^{19937} - 1$.
- PRNGs are an interesting (and fun) subject by themselves:
  - Donald E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, 3rd Edition
  - M. Matsumoto and T. Nishimura, *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator*, ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.
- A PRNG produces random floats uniformly in [0.0, 1.0).

Nota Il modulo random di numpy mi permette di generare numeri random non uno alla volta, ma in array!

Vettorizzazione

Avoid explicit for loops in Python whenever you can!

pandas: utile per leggere e scrivere file excel.

```
import random
import time
import numpy as np

# How many random numbers (uniformly distributed between 0 and 1) do you
# want to throw?
n = 1000000

# The slow way: explicit for loop in Python.
t0 = time.time()
x = []
for i in range(n):
    x.append(random.random())
dt = time.time() - t0
```

```
print('Elapsed time: %.3f s' % dt)

# The quick way: vectorizing in numpy
t0 = time.time()
x = np.random.random(size=n)
dt = time.time() - t0
print('Elapsed time: %.3f s' % dt)
```

```
[Output]
Elapsed time: 0.137 s
Elapsed time: 0.015 s
```

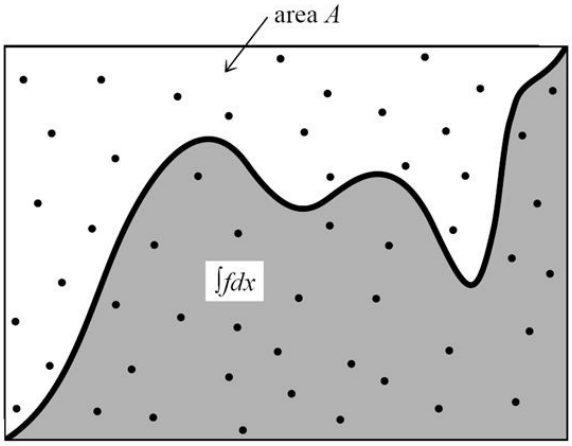
How does vectorizaion work?

- Python is know to be slooow
  - This is the price you pay for being so beautiful and flexible
- Does it matter? If depends...
  - If you are parsing a text file or fetching a web page probably not
  - If you are performing a CPU-intensive processing on a TB of data probably yes
- What's so magic in using numpy?
- numpy is written in C as a Python extension
  - Routines are highly optimized to cruch numbers
  - When you perform an array operation in Python you are actually executing optimized C code
- Basic message: avoid for loops in pure Python when crunching numbers

Secondo Assegnamento

How do I throw PRN with arbitrary pdf?

Hit or miss:

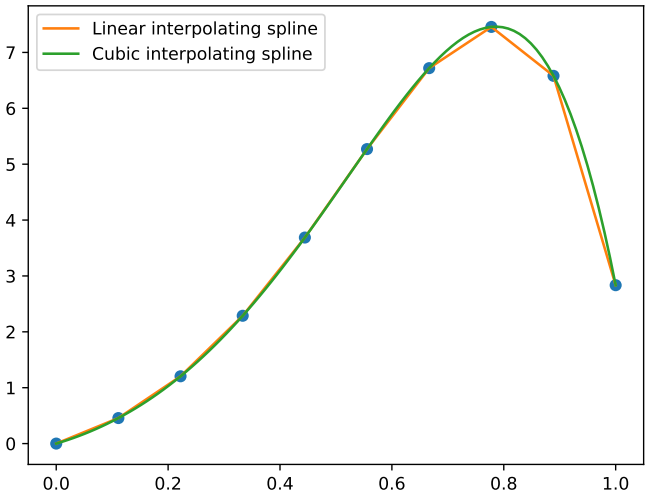


- Hit or miss, aka acceptance/rejection method:
  - Enclose your pdf in a rectangle
  - Throw a x and a y
  - Accept x if $y \leq f(x)$
- This is horrible—please don't use it!

Inverse transform

- Probability density function (pdf)
$$p(x) \quad (\geq 0)$$
- Cumulative function (cf)
$$F(x) = \int_{-\infty}^{x} p(x')dx'$$
- Percent-point function (ppf)
$$x = F^{-1}(q)$$
- Awesome fact: if q is uniformly distributed in $[0, 1]$, then $x = F^{-1}(q)$ is distributed according to $p(x)$!

An interesting object: splines



- Defined piecewsise by polinomials of degree k ($k = 3$ fairly popular)
  - Interpolating: passing through a set of pre-defined points
  - First $k - 1$ derivatives continuos at the control points
- Superior to polynomial interpolation or curve fitting in many cases

Splines: construction and properties

```
1  import numpy as np
2  from scipy.interpolate import InterpolatedUnivariateSpline
3
4  x = np.linspace(0., 1., 10)
5  y = np.exp(3. * x) * np.sin(3. * x)
6
7  s1 = InterpolatedUnivariateSpline(x, y, k=1)
8  s3 = InterpolatedUnivariateSpline(x, y, k=3)
9
10 print(s1(0.234))
11 print(s1.integral(0.2, 0.8))
```

```
1  [Output]
2  1.3192110648078448
3  2.6659857771053925
```

- Evaluation is fairly inexpensive
  - If the input x-array is sorted can do a binary search in O(log(N)) complexity
- Derivatives and integrals are easy
  - Can be calculated *exactly* by means of elementary arithmetic operations

References

- https://numpy.org/
- https://www.scipy.org/
- https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html
- https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html
- https://docs.scipy.org/doc/scipy/reference/interpolate.html

Gio 13 ott - Lezione 7

Advanced Python Features

Errors and Exceptions

- Error handling is one of the most important problem to solve when designing a program
- What should I do when I piece of code fails?
- What does fail mean?
  - Invalid input e.g. passing a path to a non existent file, or passing a string to a function for dividing numbers
  - Valid output not found, e.g searching the position of the letter 'd' in the string 'elephant'
  - Output cannot be find in a reasonable amount of time
  - Runtime resource failures: network connection down, disk space ended...
- Two phylosophies (historically):
  - Return some error flag (in different ways) to tell the user that something went wrong
  - Exceptions
- Example: a typical convention for programs is to return 0 from the main if the execution was successful and an error code (integer number) otherwise

Error flags (no)

```
# The 'find()' method for strings in python uses an error flag
text = 'elephant'
print(text.find('p')) # upon success returns the position of the substring
print(text.find('d')) # returns -1 if the substring is not found

# Why is this dangerous?
def cut_before(input_string, substring):
    """ Cut a string from the beginning up to the position before that of
    the given substring, then return it """
    pos = input_string.find(substring)
    return input_string[:(pos)]

# If the substring exists in the string everything works fine
print(cut_before('We all live in a Yellow Submarine', 'Yellow'))
# What will be the output here?
print(cut_before('We all live in a Yellow Submarine', 'Red'))
```

```
[Output]
3
-1
We all live in a
We all live in a Yellow Submarin
```

Problems of error flags

Error codes have their use (and are fine in some cases) but they suffer from a few issues:

- Choosing them is often arbitrary (and sometimes is difficult to make a sensible choice)
  - What if all the numbers can represent meaningful output of the function?

- Are cumbersome to use
  - Which error flag is used by a function? 0? -1? 99999999? → you have to go through the documentation for each!
  - If you have a deep hierarchy of functions you have to perform checks and pass the error up at every level!
- What if the caller of a function does not check the error flag?
  - The bug can propagate silently through its code!

We want something that:

- Is clearly separated from the returned output
- Cannot be silently ignored by the user
- Is easy to report to upper level without lots of lines of code

A different way

```
1  # index() is the same as find(), but rise an exception in case of failure
2  def cut_before(input_string, substring):
3      """ Cut a string from the beginning up to the position before that of
4      the given substring, then return it """
5      pos = input_string.index(substring)
6      return input_string[:(pos)]
7
8  # If the substring exists in the string everything works fine
9  print(cut_before('We all live in a Yellow Submarine', 'Yellow'))
10 # No silent bug here!
11 print(cut_before('We all live in a Yellow Submarine', 'Red'))
```

```
1  [Output]
2  We all live in a
3  Traceback (most recent call last):
4      File "snippets/exceptions_vs_err_flags.py", line 11, in <module>
5          print(cut_before('We all live in a Yellow Submarine', 'Red'))
6      File "snippets/exceptions_vs_err_flags.py", line 5, in cut_before
7      pos = input_string.index(substring)
8  ValueError: substring not found
```

La filosofia base di Python è evitare di inventare delle cose.
Come faccio ad intercettare questo value error e in quel caso a fargli fare qualcosa di specifico?

Eccezioni

- An exception is an object that can be raised (in other languages also *thrown*) by a piece of code to signal that something went wrong
- When an exception is raised the normal flow of the code is interrupted
- The program automatically propagate the exception back in the function hierarchy until it found a place where the exception is catched and handled
- If the exception is never catched, not even in the main, the program crash with a specific error message
- Cathcing the exception is done with a *try* - *except* block

Se non intercettiamo l'eccezione, il flusso del codice viene interrotto. Possiamo dire "se ho questa eccezione allora faccio questo...".

Try block

```
def cut_before(input_string, substring):
    try:
        result = input_string[:(input_string.index(substring))]
        print('This line is not executed if an exception is raised in the try block')
        return result
    # Catch the correct exception type with 'except'
    except ValueError:
        print('This line is executed only if a ValueError is raised in the try block')

cut_before('We all live in a Yellow Submarine', 'Yellow')
cut_before('We all live in a Yellow Submarine', 'Red')
```

```
[Output]
This line is not executed if an exception is raised in the try block
This line is executed only if a ValueError is raised in the try block
```

Per ogni try possiamo anche mettere più di un except.
Dobbiamo cercare di intercettare le eccezioni nel modo più specifico possibile!

`else`, `finally`

- There are two more optional statements in a try-block:
  - *else*: executed only if no exception is raised in the try block
  - *finally*: executed no matter what
- *finally* is executed even if there is a return statement in the try block
- can be used to release important resources (e.g. closing a file, or a connection)

Using `else` and `finally`

```
def cut_before(input_string, substring):
    try:
        result = input_string[:(input_string.index(substring))]
        print('This line is not executed if an exception is raised in the try block')
    except ValueError:
        print('This line is executed only if a ValueError is raised in the try block')
    else: # optional!
      print('This line is executed only if no exception is raised in the try block')
      return result
    finally: # optional!
      print('This line is always executed')

cut_before('We all live in a Yellow Submarine', 'Yellow')
cut_before('We all live in a Yellow Submarine', 'Red')
```

```
[Output]
This line
This line
This line
This line
This line
A. Manfreda (INFN)
```

```
is
is
is
is
is
not executed if an exception is raised in the try block
executed only if no exception is raised in the try block
always executed
executed only if a ValueError is raised in the try block
always executed
```

L'eccezione è un oggetto. E dentro di essa possiamo encapsulare tutte le informazioni necessarie per capire cosa è andato storto!

The beauty of exceptions

- If that was all, exceptions would only be moderately useful
- The real bargain is that you can send back information together with the exception
- In fact you *are sending a full object*: the excetpion iteslf. Surprised?
- Inside the exception you can report all kind of data useful to reconstruct the exact error, which can be used by the caller for debug or to produce meaningful error messages
- You can also select which exceptions you catch, leaving the others propagate up
- Python provides a rich hierarchy of exception classes, which you can further customize (if you want) by deriving your own subclasses

The family tree of Python exceptions

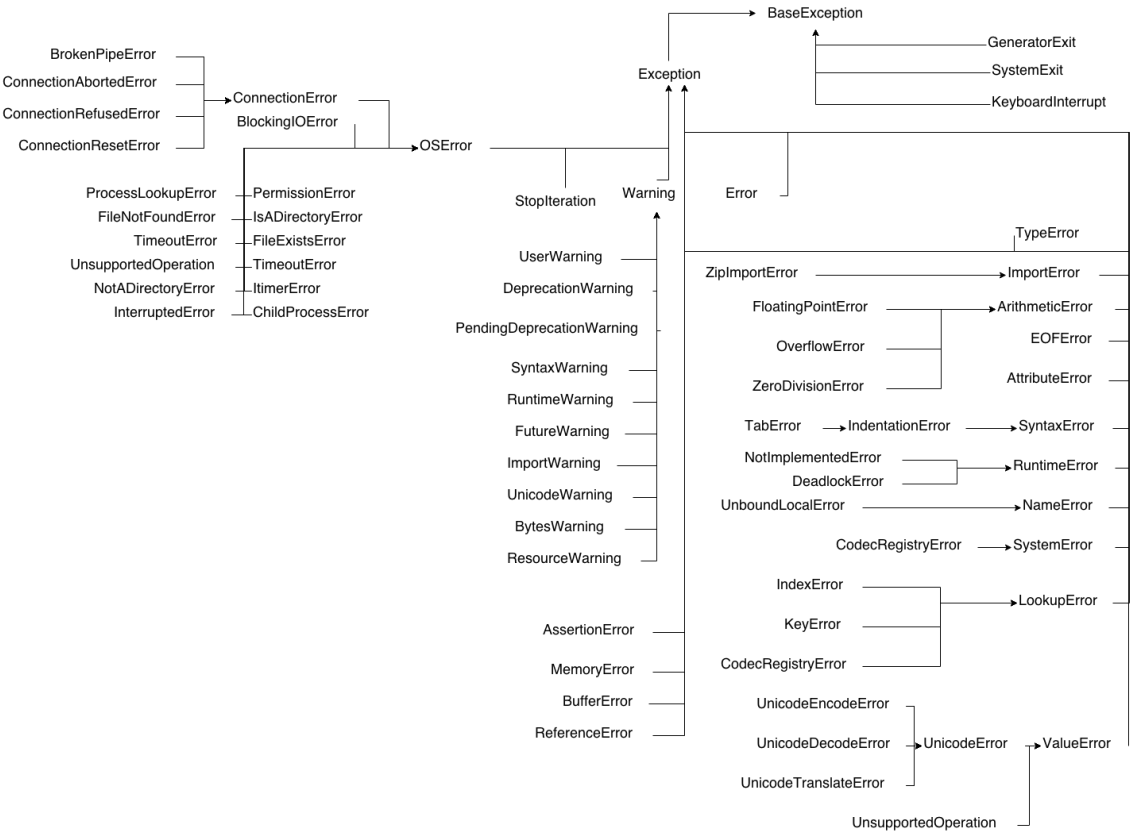


Figure 1.16: family tree

Catching specific exceptions

```
try:
    with open ('i_do_not_exist.txt') as lab_data_file:
        """ Do some process here...
        """
        pass

except FileNotFoundError as e: # we assign a name to the the exception
    print(e)

# We can be less specific by catching a parent exception
except OSError as e: # OSError is a parent class of FileNotFoundError
    print(e)

# catching Exception will catch almost everything!
except Exception as e:
    print(e)
```

```
[Output]
[Errno 2] No such file or directory: 'i_do_not_exist.txt'
```

`Exception` sta per qualsiasi altro tipo di errore.

Exception caveats

- Warning: catching *Exception*, will also catch *SyntaxError* and *NameError*

- This mean that the code will 'run' even if there is a typo in it!
- Bottom line: you should never catch generically for *Exception*, always be more specific
- Even worse, you should never catch for *BaseException* as that would even prevent the user for from aborting the execution with a *KeyboardInterrupt* (e.g. Ctrl-C)
- Unless that is what you need, of course

There is no check - only try

- In Python exceptions are the default methods for handling failures
- Many functions raise an exception when something goes wrong
- The common approach is: do not chech the input beforehand. Use it and be ready to catch exceptions if any.
- *Easier to ask for forgiveness than permission.*

Catching specific exceptions

```
1  import os
2
3  file_path = 'i_do_not_exists.txt'
4
5  # Defensive version
6  if os.path.exists(file_path):
7      # What if the file is deleted between these two lines? (by another process)
8      # What if the file exists but you don't have permission to open it?
9      data_file = open(file_path)
10 else:
11     # Do something
12     print('Oops - file \'{}\' does not exist'.format(file_path))
13
14 # Pythonic way - you should prefer this one!
15 try:
16     data_file = open(file_path)
17 except OSError as e: # Cover more problems than FileNotFoundError
18     print('Oops - cannot read the file!\n{}'.format(e))
```

```
1  [Output]
2  Oops - file 'i_do_not_exists.txt' does not exist
3  Oops - cannot read the file!
4  [Errno 2] No such file or directory: 'i_do_not_exists.txt'
```

Raising exceptions

- Up to now we have been dealing with exceptions generated by Python functions
- What about raising exceptions ourselves?

```
1  def raising_function():
2      # You can pass an useful message to the exceptions you raise
3      raise RuntimeError('this is a useful debug message')
4
5  try:
```

```
    raising_function()
except RuntimeError as e:
    # The message can be retrieved by printing the exception
    print(e)
```

```
[Output]
this is a useful debug message
```

Valutare bene prima di mettere `sys.exit('guarda che ho bisogno di quel file')`. Se invece sollevo un'eccezione, offro la scelta a chi esegue il codice.

Custom exceptions

- Beside the built-in exceptions provided by Python, you can add your own custom exceptions by inheriting from the *Exception* class
- This serves two purposes:
  - Make the exception handling code more specific, and hence more readible
  - Allows you to pass additional data with your exception - in the form of attributes of the class - which can be used for debug or any other purpose

```
class SimpleCustomError(Exception):
    pass # Yeah that's it

raise SimpleCustomError('simple error')
```

```
[Output]
Traceback (most recent call last):
File "snippets/custom_exceptions.py", line 4, in <module>
raise SimpleCustomError('simple error')
__main__.SimpleCustomError: simple error
```

altro esempio:

```
class ValueTooLargeError(ValueError):
    def __init__(self, value):
        self.value = value
        super().__init__('{}: {} is too large'.format(self.__class__.__name__,
                                                      self.value))

value = 100
try:
  if value > 10:
      raise ValueTooLargeError(value)
except ValueError as e:
    print(e)
```

```
[Output]
ValueTooLargeError: 100 is too large
```

Where to catch exceptions?

- Differently from error flags, which need to be checked as early as possible, you are not in a rush with exceptions
- Remeber: your goal is to provide the user a meaningful error message and useful debug information.
- You should catch an exception only when you have enough context to do that - which sometimes means waiting a few levels in the hierarchy!

I blocchi try-except devono essere il più piccolo possibile, e il più specifico possibile!

When to catch

```
def parse_line(line):
    """ Parse a line of the file and return the values as float"""
    values = line.strip('\n').split(' ')
    # the following two lines may generate exceptions if they fail!
    time = float(values[0])
    tension = float(values[1])
    return time, tension

with open('snippets/data/fake_measurements.txt') as lab_data_file:
    for line in lab_data_file:
        if not line.startswith('#'): # skip comments
            time, tension = parse_line(line)
            print(time, tension)
```

```
[Output]
0.1 15.2
0.2 12.4
Traceback (most recent call last):
File "snippets/when_to_catch.py", line 12, in <module>
time, tension = parse_line(line)
File "snippets/when_to_catch.py", line 6, in parse_line
tension = float(values[1])
ValueError: could not convert string to float: 'pippo'
```

Catch too early

```
def parse_line(line):
    """ Parse a line of the file and return the values as float"""
    values = line.strip('\n').split(' ')
    try:
        time = float(values[0])
        tension = float(values[1])
    except ValueError as e:
        print(e) # This is not useful - which line of the file has the error?
        return None # We can't really return something meaningful
    return time, tension

with open('snippets/data/fake_measurements.txt') as lab_data_file:
    for line in lab_data_file:
        if not line.startswith('#'): # skip comments
            time, tension = parse_line(line)
            print(time, tension) # This line still crash badly!
```

```
[Output]
0.1 15.2
0.2 12.4
could not convert string to float: 'pippo'
Traceback (most recent call last):
File "snippets/when_to_catch_1.py", line 15, in <module>
time, tension = parse_line(line)
TypeError: 'NoneType' object is not iterable
```

Catch when needed

```
def parse_line(line):
    """ Parse a line of the file and return the values as float"""
    values = line.strip('\n').split(' ')
    time = float(values[0])
    tension = float(values[1])
    return time, tension

with open('snippets/data/fake_measurements.txt') as lab_data_file:
    for line_number, line in enumerate(lab_data_file): # get the line number
        if not line.startswith('#'): # skip comments
            try:
                time, tension = parse_line(line)
                print(time, tension)
            except ValueError as e:
                print('Line {} error: {}'.format(line_number, e))
```

```
[Output]
0.1 15.2
0.2 12.4
Line 3 error: could not convert string to float: 'pippo'
0.4 13.2
```

Lun 17 ott - Lezione 8

Iterators

Quando una classe implementa il metodo `__iter__` allora diventa *iterabile*.

Iterators and iterables

Un iteratore è un oggetto definito dal fatto di sapere qual è il prossimo elemento, grazie al metodo magico `__next__`.

- An *iterable* in Python is something that has a *__iter__* method, which returns an iterator
- An *iterator* is an object that implement a *__next__* method which is used to retrieve elements one at the time
- When there are no more elements to return, the iterator signals that with a specific exception: *StopIteration()*
- An iterator also implement an *__iter__* method that return... itself. So an iterator is also technically an iterable[11]! (But the opposite is not true)

Perché passare dall'iteratore? Perché non implementare il metodo `__next__` direttamente sul nostro oggetto? Il fatto è che posso avere più iteratori attivi su uno stesso contenitore dati. Per questo non posso implementare il metodo `__next__` direttamente nella classe di dati, ma devo passare per l'iteratore.

A 'for' loop unpacked

```
1  my_list = [1., 2., 3.]
2
3  # For-loop syntax
4  for element in my_list:
5      print(element)
6
7  # This is equivalent (but much less readible and compact)
8  list_iterator = iter(my_list)
9  while True:
10     try:
11         print(next(list_iterator))
12     except StopIteration:
13         break
```

Salvo il mio iteratore in una variabile e inizio un ciclo (potenzialmente infinito). Quando l'iterazione solleva l'eccezione `StopIteration` interrompo il ciclo.

```
1  [Output]
2  1.0
3  2.0
4  3.0
5  1.0
6  2.0
7  3.0
```

[11] Only 'technically' because an iterator has no data of its own, so you always need a 'real' iterable to actually iterate

A simple iterator

```
class SimpleIterator:
    """ Class implementing a super naive iterator"""

    def __init__(self, container):
        self._container = container
        self.index = 0

    def __next__(self):
        try:
            # Note: here we are calling the __getitem__ method of self._container
            item = self._container[self.index]
        except IndexError:
            raise StopIteration
        self.index += 1
        return item

    def __iter__(self):
        return self

class SimpleIterable:
    """ A very basic iterable """

    def __init__(self, *elements):
        # We use a list to store elements internally.
        # This provide us with the __getitem__ function
        self._elements = list(elements)

    def __iter__(self):
        return SimpleIterator(self._elements)
```

Nel costruttore gli passo il contenitore di dati su cui voglio iterate. Mi salvo una referenza a questo contenitore dati. E faccio partire l'indice da zero. Nota: questo funziona per le liste, tuple e array, ma non per i dizionari, che non restituiscono `indexError`, ma `KeyError`!

```
from simple_iterator import SimpleIterable

my_iterable = SimpleIterable(1., 2., 3., 'stella')
for element in my_iterable:
    print(element)
```

```
[Output]
1.0
2.0
3.0
stella
```

A crazy iterator

```
import random

class CrazyIterator:
    """ Class implementing a crazy iterator"""

    def __init__(self, container):
        random.seed(1)
```

```
        self._container = container

    def __next__(self):
        try:
            # We get one possibility out of len(self._container) to exit
            index = random.randint(0, len(self._container))
            item = self._container[index]
        except IndexError:
            raise StopIteration
        return item

    def __iter__(self):
        return self

class CrazyIterable:
    """ Similar to a simple iterable, but with a twist... """

    def __init__(self, *elements):
        self._elements = list(elements)

    def __iter__(self):
        return CrazyIterator(self._elements)
```

```
from crazy_iterator import CrazyIterable

my_iterable = CrazyIterable('A', 'B', 'C', 'D', 'E')
for element in my_iterable:
    print(element)
```

```
[Output]
B
E
A
C
A
D
D
D
```

Python tools for iterables

- Python provides a number of functions that consume an iterable and return a single value:
  - `sum`: Sum all the elements
  - `all`: Return true if a given condition is true for all the elements
  - `any`: Return true if a given condition is true for at lest one element
  - `max`: Return the max
  - `min`: Return the minimum
  - `functools.reduce`: Apply a function recursively to pairs of elements

Generatori

Gli iteratori operano su dati **esistenti**. Tuttavia, a volte vorremmo iterare su qualcosa che non esiste già da prima. Ad esempio, se vogliamo generare in maniera iterativa tutti i numeri della serie di Fibonacci; ad ogni iterazione vogliamo generare il prossimo.
Questa cosa non si può fare con gli iteratori, ma si fa con i **generatori**:

- We have seen that iterators are useful to iterate over container
- However that assumes a containers exists → memory usage
- Generators allow you to loop over sequences of items even when they don't exist before - the items are just created lazily the moment they are required (**lazy:** una cosa che viene fatta all'ultimo momento possibile.)
- For example you can write a generator to loops over the Fibonacci succession. You can't create the sequence earlier, since it is not finite!
- Generators are created through either generator expressions or generator functions
- In real life most of the time you will simply use pre-made functions that return a generator, like *range()* (in Python 3)
- Generator can be used to iterate in for loops, just like iterators

Generators first look

Sui generatori possiamo iterare esattamente come sugli iteratori: la sintassi è la stessa.

```
""" range() is a function that returns a generator in Python 3. The list of
numbers never exists entirely, they are created ine at a time.
Note: In Python 2 range() does create the full list at the beginning.
There used to be a xrange() function for lazy generation, which is now
deprecated in Python 3. """
for i in range(4):  # generators act like iterators in for loop
    print(i)

data = [12, -1, 5]
square_data_generator = (x**2 for x in data) # generator expression!
for square_datum in square_data_generator: # again, works like an iterator
    print(square_datum)
```

```
[Output]
0
1
2
3
144
1
25
```

Generator functions

- A generator function is a function that contains the keyword yield at least once in his body
- When you call a generator function the code is not executed - instead a generator object is created and returned (even if you don't have a return statement)
- Each call to *next()* on the returned generator will make the function code run until it finds a *yield* statement
- Then the execution is paused and the value of the expression on the right of *yield* is returned (yielded) to the caller
- A further call of next will resume the execution from where it was suspended until the next *yield* and so on

- Eventually, when the function body ends, *StopIteration* is raised
- Usually generators functions contain a loop - but it's not mandatory!

Quando noi chiamiamo una funzione generatrice, non viene eseguito il corpo della funzione, bensì viene restituito un generatore.

```
def generator_function_simple():
    print('First call')
    yield 1
    print('Second call')
    yield 2
    print('I am about to rise a StopIteration exception...')

gen = generator_function_simple() # A generator function returns a generator
print(next(gen)) # We stop at the first yield and get the value
print(next(gen)) # Second yield
next(gen) # The third next() will throw StopIteration
```

```
[Output]
First call
1
Second call
2
I am about to rise a StopIteration exception...
Traceback (most recent call last):
    File "snippets/generator_functions.py", line 11, in <module>
        next(gen) # The third next() will throw StopIteration
StopIteration
```

Infinite sequence generators

Tipicamente all'interno del generatore c'è un loop.

```
# Generator function that provides infinte fibonacci numbers
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# We need to impose a stop condition externally to use it
max_n = 7
fib_numbers = []
for i, fib in enumerate(fibonacci()):
    if i >= max_n:
        break
    else:
        fib_numbers.append(fib)
print(fib_numbers)

# Another way of doing that is using 'islice' from itertools
import itertools
# Generator expression
fib_gen = (fib for fib in itertools.islice(fibonacci(), max_n))
print(list(fib_gen))
```

```
[Output]
[0, 1, 1, 2, 3, 5, 8]
[0, 1, 1, 2, 3, 5, 8]
```

islice prende un certo numero di elementi da un iteratore.

Un generatore serve in tutti quei casi in cui voglio generare i vari elementi in maniera lazy.

Python generator functions

- Python provides a number of built-in functions that return a generator from an iterable, such as:
  - `enumerate`: Automatic counting of iterations
  - `map`: Apply a function to the elements
  - `filter`: Return only the elements passing a given condition
  - `zip`: Return pairs of elements (requires two sequences)
  - `reversed`: Loop in the reversed order
- Countless others can be found in the `itertools` library
  - `islice`: Slice the loop with start, stop and step
  - `takewhile`: Stop looping when a condition becomes false
  - `accumulate`: Get the results of applying the function iteratively to pair of elements
  - `chain`: Loop through many sequences one after another
  - `cycle`: Loop over the sequence repeatedly, indefinitely
  - `permutations`: Get all the permutations of a given length
  - `product`: Compute the cartesian product of iterables
  - `groupby`: Group by value of some key (function)
  - And so on...
- Take a look at the documentation of each function to see how to properly call it!

Itertools showcase

```python
from itertools import accumulate, product, chain, groupby, permutations, combinations
import operator

l1 = [1, 2, 3, 4]
print(list(accumulate(l1)))
print(list(accumulate(l1, func=operator.mul)))
print(list(combinations(l1, 3)))

l2 = [5, 6]
print(list(permutations(l2, 2)))
print(list(product(l1, l2)))

def is_even(n):
    return n % 2 == 0

l3 = list(chain(l1, l2))
# groupby expect the list to be sorted by the grouping function
l3.sort(key=is_even)
for k, g in groupby(l3, key=is_even):
    print(k, list(g))
```

```
[Output]
[1, 3, 6, 10]
[1, 2, 6, 24]
[(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4)]
[(5, 6), (6, 5)]
[(1, 5), (1, 6), (2, 5), (2, 6), (3, 5), (3, 6), (4, 5), (4, 6)]
False [1, 3, 5]
True [2, 4, 6]
```

Lambda functions

Sono un modo per creare una funzione anonima (senza un nome).

- Anonymous functions, or lambda functions are a construct typical of functional programming
- `https://en.wikipedia.org/wiki/Lambda_calculus`
- `https://en.wikipedia.org/wiki/Functional_programming`
- In Python a lambda function is essentially a special sintax for creating a function on the fly, without giving it a name
- They are limited to a single expression, which is returned to the user
- Many of the typical uses for lambdas are already covered in python by generator expressions and comprehension, so this is more like a niche feature of the language

```
# Here we create a lambda function and assign a name to it (ironically)
multiply = lambda x, y: x * y
# Use it
print(multiply(5, -1))

# Typical use is inside generator expressions
numbers = range(10)
squares = list(map(lambda n: n**2, numbers))
print(squares)

# However, remeber that you can do the same with list comprehension
squares = [n**2 for n in numbers]
print(squares)
```

Il corpo di una funzione deve essere solo una riga.
`lambda argomenti:  output`

```
[Output]
-5
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Recap example: file iterator

```
from itertools import dropwhile

class LabFileIterator:
    def __init__(self, file_obj):
        self._lines = dropwhile(lambda line: line.startswith('#'), file_obj)

```

```
    def __next__(self):
        line = next(self._lines)
        values = line.strip('\n').split(' ')
        time = float(values[0])
        tension = float(values[1])
        return time, tension

    def __iter__(self):
        return self

with open('snippets/data/fake_measurements.txt') as lab_data_file:
    try:
        for line_number, (time, tension) in enumerate(LabFileIterator(lab_data_file)):
            print(line_number, time, tension)
    except ValueError as e:
        # Here we get the wrong line number! Why?
        print('Line {} error: {}'.format(line_number, e))
```

```
[Output]
0 0.1 15.2
1 0.2 12.4
Line 1 error: could not convert string to float: 'pippo'
```

File iterator redone

```
from itertools import dropwhile

class LabFile:
    def __init__(self, file_obj):
        self._file = file_obj

    def __iter__(self):
        # Enumerate is now inside dropwhile, so all lines are counted
        # This is a bit convoluted, though
        for i, line in dropwhile(lambda x : x[1].startswith('#'),
                                 enumerate(self._file)):
            values = line.strip('\n').split(' ')
            try:
                time = float(values[0])
                tension = float(values[1])
            except ValueError as e:
                print('Line {} error: {}'.format(i, e))
                continue
            yield time, tension

with open('snippets/data/fake_measurements.txt') as lab_data_file:
    for time, tension in LabFile(lab_data_file):
        print(time, tension)
```

```
[Output]
0.1 15.2
0.2 12.4
Line 3 error: could not convert string to float: 'pippo'
0.4 13.2
```

File iterator, final version

```
class LabFile:
    def __init__(self, file_obj):
        self._file = file_obj

    def __iter__(self):
        # This is more readible
        for i, line in enumerate(self._file):
            if line.startswith('#'):
                continue
            values = line.strip('\n').split(' ')
            try:
                time = float(values[0])
                tension = float(values[1])
            except ValueError as e:
                print('Line {} error: {}'.format(i, e))
                continue
            yield time, tension

with open('snippets/data/fake_measurements.txt') as lab_data_file:
    for time, tension in LabFile(lab_data_file):
        print(time, tension)
```

```
[Output]
0.1 15.2
0.2 12.4
Line 3 error: could not convert string to float: 'pippo'
0.4 13.2
```

Decorators

non ha avuto tempo di farli tutti, vediamo solo:

The @classmethod decorator

costruttore alternativo

- We have already seen a built-in Python decorator: *@property*
- We used that to get proper encapsulation
- There is another built-in decorator one which is very useful for classes: *@classmethod*
- A classmethod is like a class attribute: you don't need an instance to use it
- A class method can access class attributes but not instance attributes
- The main use for class methods is to provide alternate constructors

```
import numpy

class LabData:

  def __init__(self, times, values):
     """ Our usual constructor"""
     self.times = numpy.array(times, dtype=numpy.float64)
     self.values = numpy.array(values, dtype=numpy.float64)

```

```
  @classmethod # The classmethod decorator
  def from_file(cls, file_path): # We get the class as first argument, not self
      """ Constructor from a file"""
      print(cls)
      times, values = numpy.loadtxt(file_path, unpack=True)
      # We call the constructor of 'cls' which is our LabData
      # This is not a 'real' constructor, we need to return the object!
      return cls(times, values)

# We call the alternate constructor from the class itself, not from an instance!
lab_data = LabData.from_file('snippets/data/measurements.txt')
print(lab_data.values)
```

```
[Output]
<class '__main__.LabData'>
[15.2 12.4 11.7 13.2]
```

Chapter 2

Parallel Computing

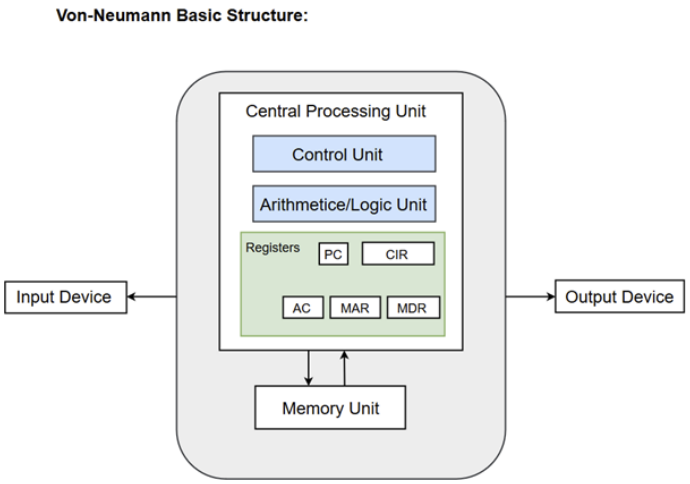
Gio 20 ott - Lezione 9

Computer architecture from a performance point of view: from serial to parallel

Architettura di Von Neumann

L'architettura di Von Neumann è una tipologia di architettura hardware per computer digitali programmabili a programma memorizzato la quale condivide i dati del programma e le istruzioni del programma nello stesso spazio di memoria, contrapponendosi all'architettura Harvard nella quale invece i dati del programma e le istruzioni del programma sono memorizzati in spazi di memoria distinti. Introdotta nel 1945 da John Von Neumann, consiste di 5 elementi:

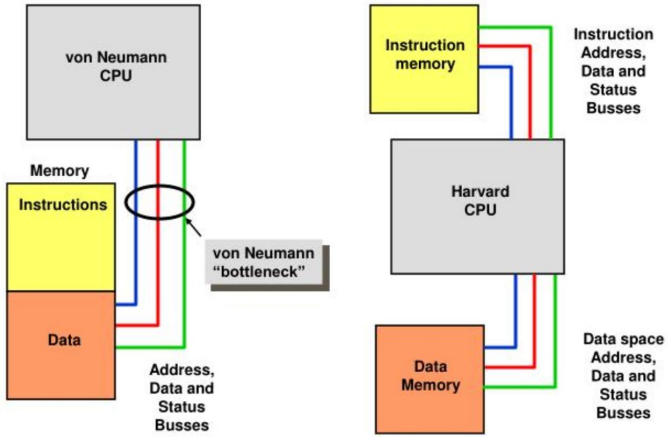
1. Processing unit (arithmetic logic unit)
2. Control unit (instruction pool)
3. Memory
4. Bus
5. I/O



Da una parte abbiamo i dati, dall'altra abbiamo i programmi. La parte di controllo copia i dati dalla memoria in una memoria temporanea e vi esegue i comandi contenuti nei programmi.

Von Neumann Bottleneck

L'architettura di Von Neumann presenta delle limitazioni legate al fatto che viene condiviso lo stesso bus per dati e istruzioni, creando il cosiddetto *Von Neumann Bottleneck*.



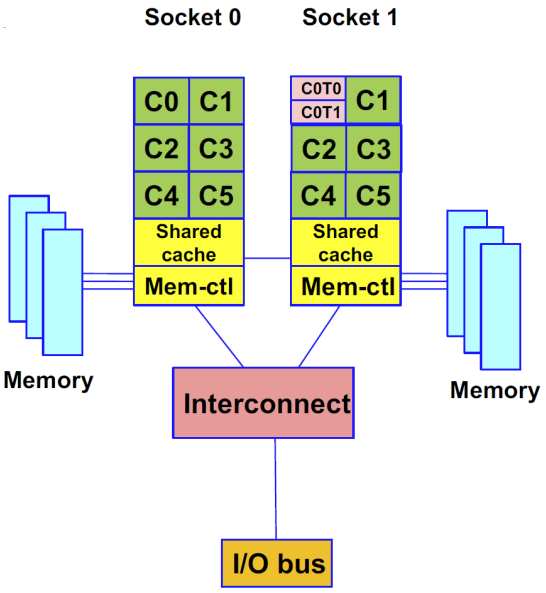
Esistono varie strategie per mitigare questo fenomeno:

1. Caching and memory gerarchy on chip
2. Separate access to data and instructions (Harvard Architecture)
3. Branch prediction

Simple Server architecture

In a server multiple components interacts during the program execution.

- Processors/cores
  - I-cache, D-cache
- Shared Caches
  - For instruction and data
- Memory controllers
- I/O subsystems
  - Storage, network, peripherals



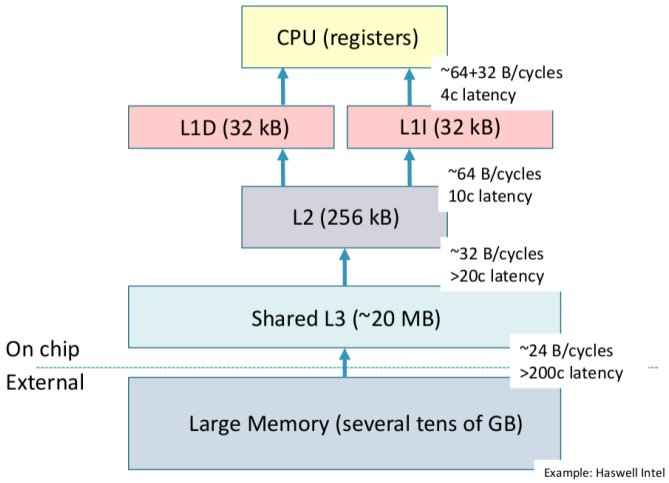
An example: NUMA architecture (non- uniform memory access).

Memoria

Ci sono 2 parametri che caratterizzano le memorie:
Banda: numero di byte che posso estrarre dalla memoria ad ogni colpo di clock.
Latenza: quanto tempo ci vuole dopo che abbiamo richiesto i dati ad ottenerli effettivamente.

Se ho un'operazione che eseguo molto spesso, non conviene ogni volta accedere a questa operazione. Analogamente, se abbiamo gli stessi dati su cui fare delle operazioni, li carichiamo nella cache una volta sola e poi facciamo le operazioni.

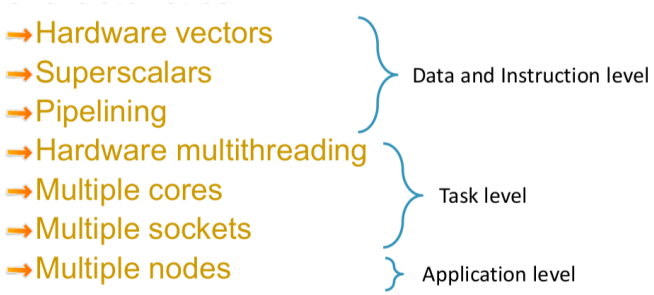
In particolare, la cache è strutturata su più livelli, ognuno dei quali ha performance diverse in termini di Banda e Latenza.
La gerarchia di "data access" e "instruction fetching" è fondamentale nell'architettura dei computer.



Più il clock va veloce e più il processore è veloce. Tuttavia la velocità del clock non può aumentare all'infinito. Si cercano metodi per andare più veloci del tempo scandito dal clock.

Seven dimensions of performance

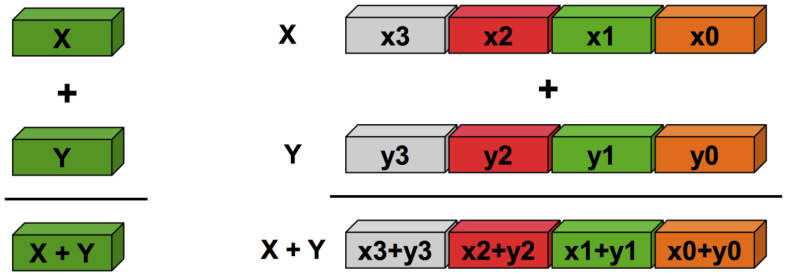
The «modern» PC performance depends on (at least) seven characteristics:



Processori Vettoriali

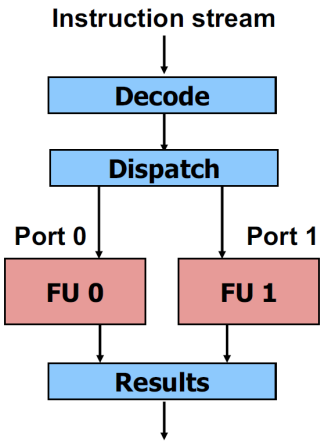
Finora abbiamo visto processori "scalari".
Modern processors implement registers for vectorialization (SSE/SSE2 and AVX)

- Scalar mode:
  - One operation produces one result
- SIMD (Single Instruction Multiple Data) is a simple way to parallelize
  - One operation produces multiple results



Superscalari

Abbiamo tanti processori scalari, ognuno dei quali fa singole operazioni su singoli elementi di memoria.

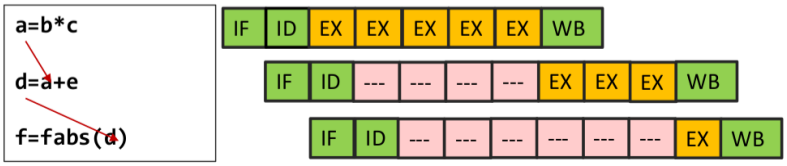


- Architecture between pure «scalar» and pure «vector»
  - Several hardware units can execute different operation on different data at the same time
- Functional Units (FU) can have identical or different computing capabilities
  - Decoder and Dispatcher must have the capability to manage two instruction in one clock cicle
- Usefull for Branch Prediction
  - Execute at the same time different branches in an algorithm then choose the correct one

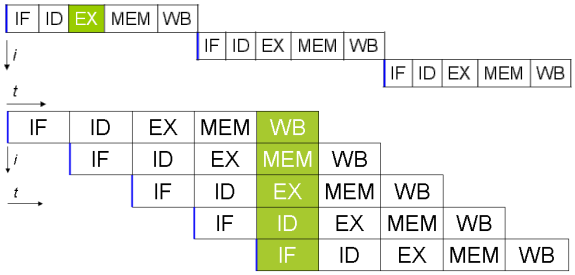
Branch prediction: Ho sufficienti risorse per eseguire contemporaneamente varie branch di un programma.

Pipelining

Pipelining consists in the capability to execute different stage of consecutive instructions at the same time.



The pipeline is an important ingredient in modern processors. However, it isn't always possible to fully exploit the pipeline.



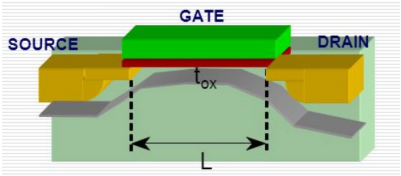
Summary:

- Superscalars, Pipelining and Vectorialization are methods to exploit some «parallelism» at the instruction and data level: ILP
  - Probably OOO (Out-of-order) execution should be included in this category
- The possible improvement thanks to ILP depends on problem and data structures
  - 1x-10x for Superscalars and Pipeline
  - 2x,4x,8x,16x for the vectorialization
- These methods show «saturation» because they are limited by the CPU resources available
  - Pentium 4: 30 pipeline stages (nowadays 10-15 maximum)
  - ARM A57 (Apple A7/A8): 9 ports/6 instructions superscalar
  - Intel Tiger Lake: vector of 512 bits for a subset of AVX512 instructions

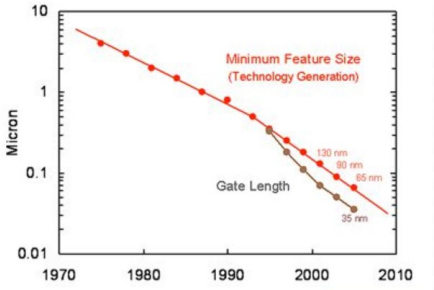
... the point is: can CPU resources grow indefinitely?

Dennard Scaling

Aka MOSFET scaling (Dennard scaling after an article from Dennard et al. in 1974 in IEEE Journal of Solid State Circuits)
-In each generation of CMOS based IC the power consumption remains the same
Breakdown of Dennard scaling around 2006: With very small integration it is not true anymore that the power consumption is the same, due to increasing in current leakage. The increasing of the speed of the transistors switching (frequency) is not anymore linear with the performance of the CPU



Energy consumption has become more important to users (For mobile, IoT, and for large clouds).
Processors have reached their power limit: Thermal dissipation is maxed out (chips turn off to avoid overheating!). Even with better packaging: heat and battery are limits.

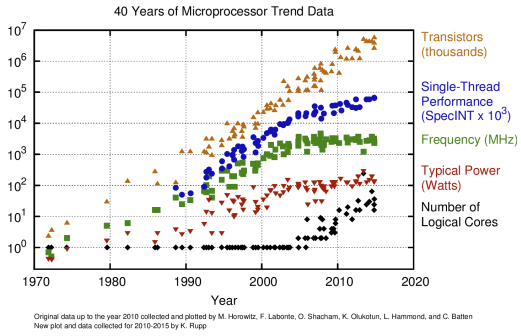


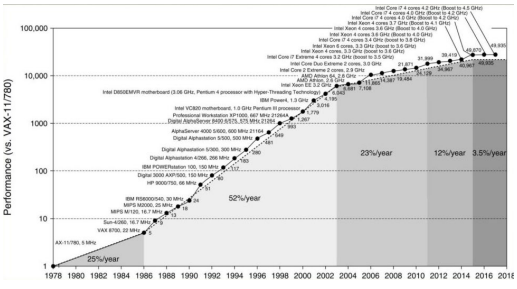
Device or Circuit Parameter	*Scaling Factor*
Device dimension t_{ox}, L, W	$1/k$
Doping concentration N_a	k
Voltage V	$1/k$
Current I	$1/k$
Capacitance eA/t	$1/k$
Delay time per circuit VC/I	$1/k$
Power dissipation per circuit VI	$1/k^2$
Power density VI/A	1

Table I: Scaling Results for Circuit Performance (from Dennard)

Moore scaling

Moore's «law» is the empirical observation that the number of transistors doubles about each two years (the performance of CPU doubles each 18 months).
Moore's prediction was verified for decades, however, around 2005 is starts to show saturation!
Moore's law is closely related to Dennard scaling.





Hardware parallelism

How to avoid saturation?
Instruction level parallelism achieved significant performance advantages.But the performance are related to clock speed. Increasing in ILP is still possibile but the complexity of CPU is more than linear, diminishing return in efficiency.
We need a next level in parallelism!

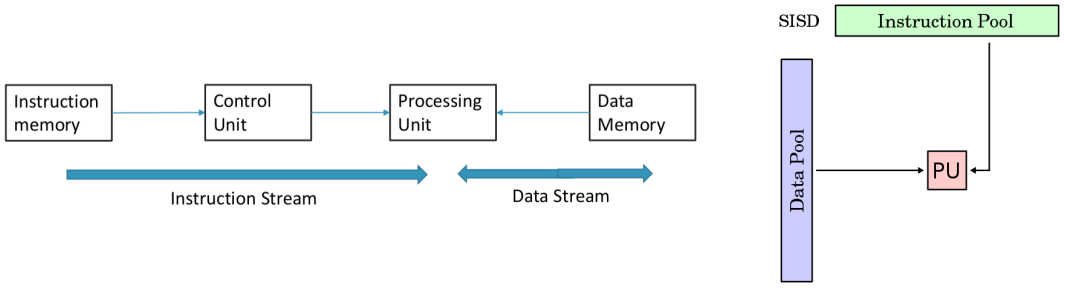
Flynn's taxonomy

Classification of computers architectures based on the number of data streams and instructions streams.

- Single Instruction Single Data (SISD): Traditional sequential computing
- Singe Instruction Multiple Data (SIMD)
- Multiple Instructions Single Data (MISD)
- Multiple Instructions Multiple Data (MIMD)

SISD: Single Instruction Single Data

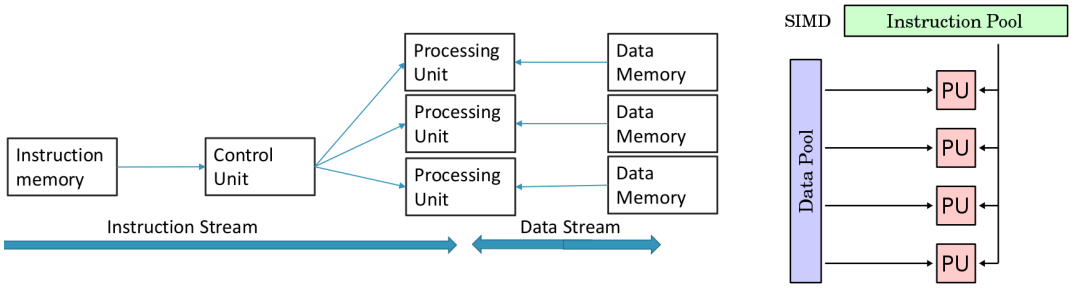
Only one instruction operates for each time slot on one data (sequential processing).



SIMD: Single Instruction Multiple Data

At one time one instruction operates on multiple data.

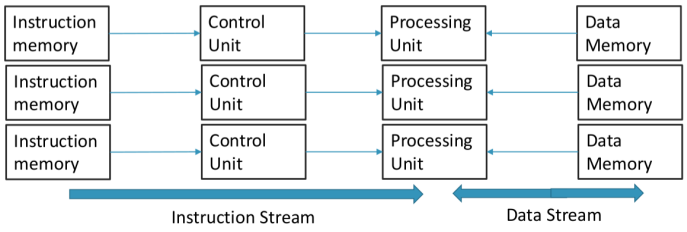
- Very similar to vector processors (although in the vector architecture the parallelism is obtained with a pipeline, while in SIMD the operations are really parallel on vector's element.)
- Array processors
- Most modern processors contain one or more SIMD sections

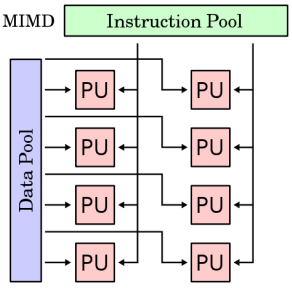


MIMD: Multiple Instruction Multiple Data

Multiple instructions streams operate on multiple data stream.

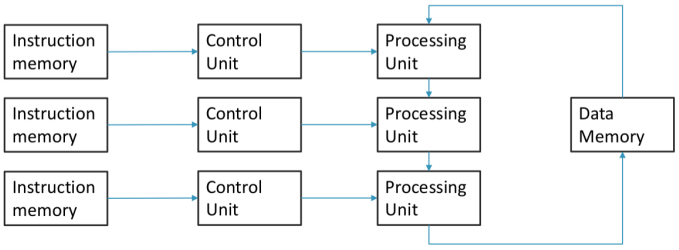
- Most of supercomputers are organized as MIMD architecture
- Multi-core superscalar, multi-processors and distributed systems

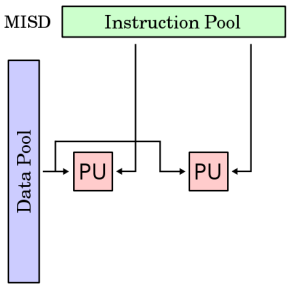




MISD: Multiple Instruction Single Data

Not commonly seen. Sometime the systolic array is seen as MISD.
Usually is an architecture used for fault tollerance and not for computing.

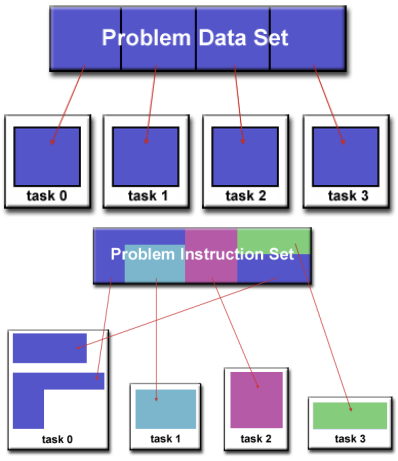




Logic partitioning and decomposition

The choice of the architecture depends on the problem.

- Domain decomposition
  - Single program, multiple data
  - decomposition based on Input domain, output domain, both
- Functional decomposition
  - Multiple programs, multiple data
  - Independent tasks
  - Pipeling



Ad esempio, se devo fare il prodotto tra matrici, divido le matrici in blocchi e faccio il prodotto.

Multiprocessor Execution Model

A specific architecture is suitable for a specific problem, but all needs «multiprocessors». Examples:

- Each processor has its own PC and executes an independent stream of instructions (MIMD)

- Different processors can access the same memory space
- Processors can communicate via shared memory by storing/loading to/from common locations

Two ways to use a multiprocessor:

- Deliver high throughput for independent jobs via job-level parallelism
- Improve the run time of a single program that has been specially designed to run on a multiprocessor - a parallel-processing program

Sequential processing

Only one "thread" of execution:
-One step follows another in sequence
-One processor is all that is needed to run the algorithm

Thread definition: It is the smallest of a program that can be managed independently by a scheduler (typically in the operating system).

- A thread is a component of a process
- Multiple threads can exist within one process
- Systems with a single processor generally implement multithreading by time slicing (software threads)

Concurrent Processing

A system in which:
-Multiple tasks can be executed at the same time
-The tasks may be duplicates of each other, or distinct tasks
-The overall time to perform the series of tasks is reduced

Advantages:
-Concurrent processes can reduce duplication.
-The overall runtime of the algorithm can be significantly reduced.
-More real-world problems can be solved than with sequential algorithms alone.

Disadvantages
-Runtime is not always reduced, so careful planning is require
-Concurrent algorithms can be more complex than sequential algorithms
-Shared data can be corrupted
-Communication between tasks is needed

Types of concurrent processing:

-Multiprogramming
-Multiprocessing
-Multitasking
-Distributed Systems

Multiprogramming

-Share a single CPU among many users or tasks.
-May have a time-shared algorithm or a priority algorithm for determining which task to run next
-Gives the illusion of simultaneous processing through rapid swapping of tasks (interleaving).

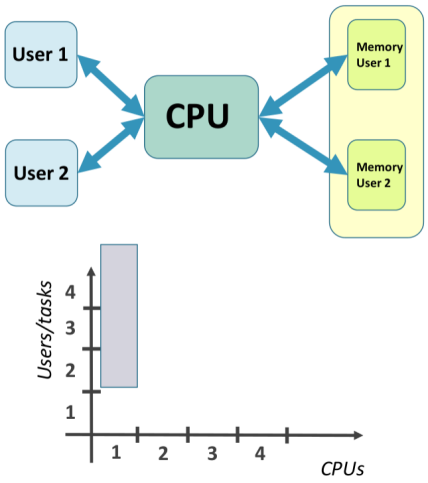


Figure 2.6: Multiprogramming

Multiprocessing

-Executes multiple tasks at the same time
-Uses multiple processors to accomplish the tasks
-Each processor may also timeshare among several tasks
-Has a shared memory that is used by all the tasks

Multitasking

-A single user can have multiple tasks running at the same time.
-Can be done with one or more processors.
-Used to be rare and for only expensive multiprocessing systems, but now most modern operating systems can do it.

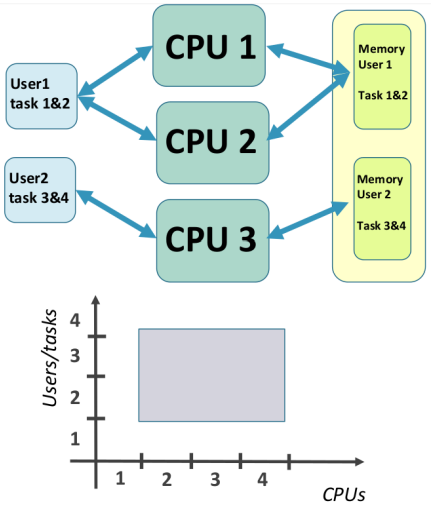
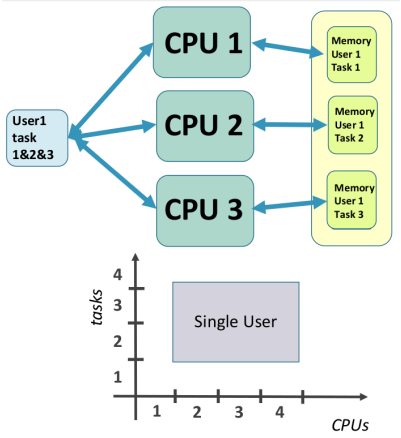


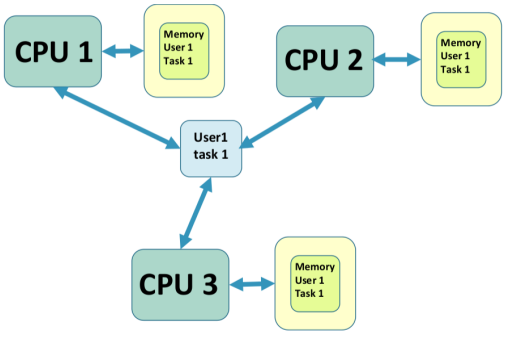
Figure 2.7: Multiprocessing

Distributed systems

-Multiple computers working together with no central program "in charge."
-No bottlenecks from sharing processors
-No central point of failure
-Complexity
-Communication overhead
-Distributed control



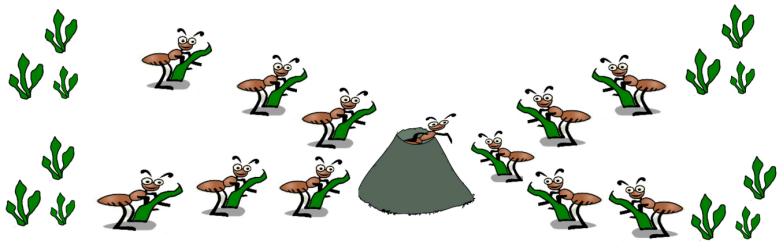
(a) Multitasking



(b) Distributed Systems

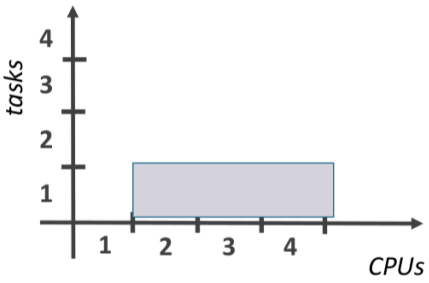
Parallelism vs Concurrency

Concurrency is the execution of multiple tasks at the same time, regardless of the number of processors.
Parallelism is the execution on multiple processors on the same task: -Breaking the task into meaningful pieces
-Doing the work on many processors
-Coordinating and putting the pieces back together.



Parallelization

For a wide class of algorithms parallelization is the most powerfull way to decrease execution time (not complexity).
-Example: a problem with O(NlogN) complexity (for instance Quicksort) on logN processors will take the time needed by O(N) algorithms
-Example: a problem with $O(N^2)$ complexity (for instance binary search) on N processors will take the time needed by O(N) algorithms



Parallelization is not free. Processors must be controlled and coordinated. We need a way to govern which processor does what work; this involves extra work.

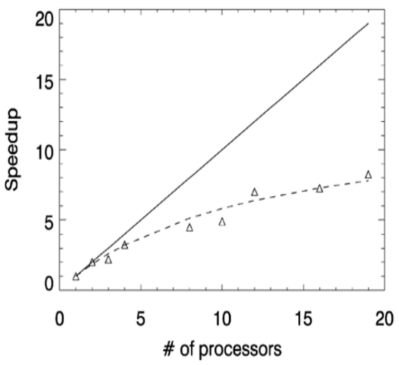
Often the program must be written in a special programming language for parallel systems. Often, a parallelized program for one machine (with, say, 2K processors) is not optimal on other machines (with, say, 2L processors).

Speedup and Efficiency

How much gain can we get from parallelizing an algorithm?
Let's define the «speedup» as (where n is the number of processors):

$$S_n = T_{serial}/T_{parallel}(n)$$

For a perfect parallel algorithm $S_n = n$. That's pratically impossible, even if for very specific cases could be also $S_n > n$ (superlinear case).



The efficiency is defined as:

$$E = S_n/n \qquad (2.1)$$

It is a measure of how well our algorithm is using the processors.

Cost and Scalability

Cost: the number of CPU required

$$c = nT_p(n) = \frac{T_1}{E}$$

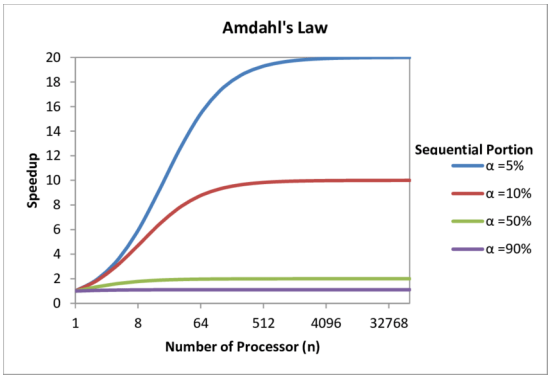
Scalability: capability to remain efficient with the increasing of the number of processors.

Amdhal's law (1967)

If only one part (P_K) of the code can be improved, the maximum improvement is given by:

$$1/\sum \frac{P_K}{S_K}$$

Where k is the part of the code and S_k is the speedup of the part-k.



In the case of parallel programming:

$$S_n = \frac{n}{nF + (1 - F)}$$

if $n \longrightarrow \infty$ the speedup is $S_n = 1/F$
For instance if the fraction of serial code is 10% (F=0.10) the maximum speedup is «only» 10 (regardless the number of processors).
Apparently the parallelism is usefull only for «embarassingly parallel» problems, with a small number of processors.

Overhead of parallelization

Load balancing

In case of several tasks in parallel the execution time of each task must be similar. Otherwise the total time is dominated by the slower task.
Some processor could be inefficently IDLE. It's not easy to design a priori a good load balancing.

Synchronization

If the tasks use the same memory (shared memory) to exchange data a logic of lock-unlock must be designed. This involves a waste of time.

Comunication latency

If data must be moved between processors the overhead due to data transmission can be really relevant.

Limits of Amdhal's law

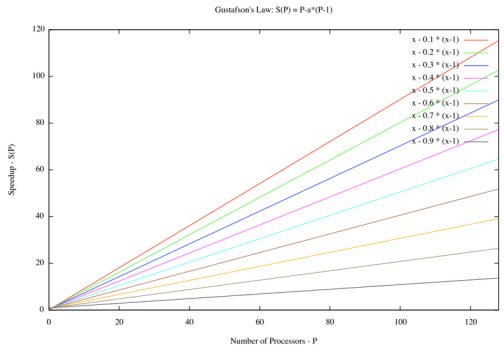
Apparently the Amdahl's law puts important limits to the advantages of parallel computing. But there are importants caveat to this law:

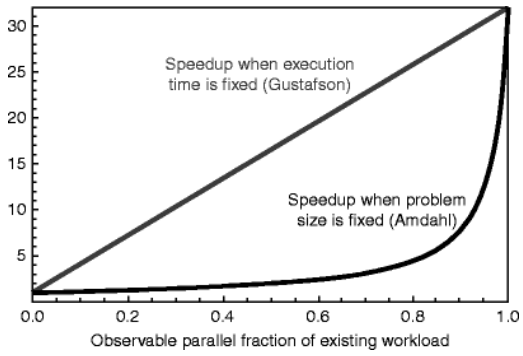
-Amdahl assumes that the best solution is always the best serial algorithm. Often some problem must be solved in parallel
-Some architectural design can help parallel processing (for instance the caching)
-Amdahl assumes that the dimension of the problem is always the same with the increasing of the number of processors. But more processors often means that wider problem can be addressed.

Gustafson's law (1988)

Let's assume s is the time of the serial part (and p is the time parallel part).
Let's assume that the problem grows with the number of processor (N) and that the serial part remains always the same.
Under these assumptions the speedup is given by:

$$s_n = N + (1 - N)s$$





The speedup is linear with N.

Recap:

Standard processors are designed for "sequential" programming

-Several "tricks" are applied at instruction level to better exploit the Von Neuman structure (Vector processors, superscalars, pipeline, ...)

-Starting from about 2005 the performances serial processors start to show saturation (Moore's law, Denard's scaling)

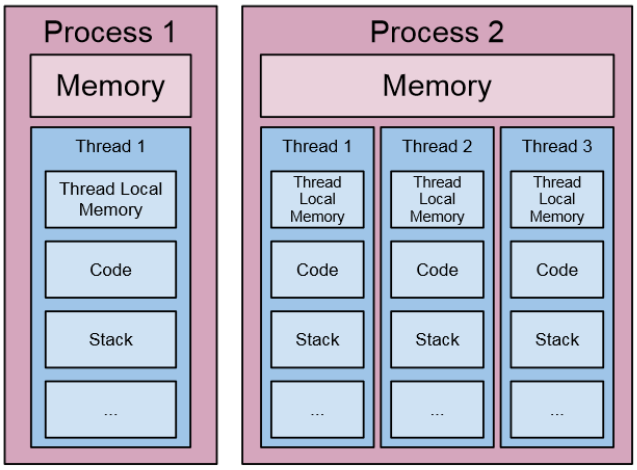
-To overcome these limitations it is necessary to rethink the way of programming (Concurrency & Parallelism)

-The idea: divide the problem in sub-problems to be addressed simultaneously (different architectures for parallelism: Flynn's taxonomy)

Multithreading and multiprocessing in Python

Threads and processes

Threads and processes are the way to use concurrency in python.
Python implements a very simple thread-safe mechanism: Global Interpreter Lock (GIL). In order to prevent conflicts only one statement in one thread is executed at a time (single-threading).



The Global Interpreter Lock (GIL)

The Global Interpreter Lock refers to the fact that the Python interpreter is not thread safe. There is a global lock that the current thread holds to safely access Python objects. Because only one thread can acquire Python Objects/C API, the interpreter regularly releases and reacquires the lock every 100 bytecode of instructions. The frequency at which the interpreter checks for thread switching is controlled by the `sys.setcheckinterval()` function. In addition, the lock is released and reacquired around potentially blocking I/O operations.
It is important to note that, because of the GIL, the CPU-bound applications won't be helped by threads. In Python, it is recommended to either use processes, or create a mixture of processes and threads.

Processi e Thread

Il processo è un'istanza del programma che abbiamo scritto. Ogni processo ha una memoria dedicata.
Quando due processi vengono lanciati, le due memorie non si parlano tra loro, sono completamente separate. Ogni processo ha una memoria chiusa.

All'interno di un singolo processo possiamo creare task differenti. Queste task possono essere viste come parti diverse del programma eseguite in modo seriale (ad es quando definiamo più funzioni che fanno compiti differenti per rendere più leggibile il programma).
Possiamo rendere questi task dei *thread*: pezzi di codice che runna indipendentemente dagli altri. C'è una memoria comune che è la memoria del processo. Poi ci sono thread diversi che runnano su risorse differenti (o sulla stessa risorsa) contemporaneamente.

Qualche volta è necessario che questi thread che stanno lavorando insieme, comunichino tra di loro. Magari vogliono leggere o scrivere qualcosa sulla memoria condivisa.Serve un meccanismo di comunicazione tra i vari thread. In che modo farli comunicare dipende da noi.

Il sistema operativo mette a disposizione due modi per mettere in comunicazione i thread, cercando di evitare possibili conflitti. **Mutex**: è un sistema di locking: quando un thread vuole accedere a una parte di memoria o a una risorsa hardware, dice "questo lo sto usando io" e gli altri

thread devono mettersi in coda fino a quando il lock non viene sganciato.
Meccanismo dei Semafori: si basa sul fatto che un thread comunichi agli altri thread cosa sta facendo. Nel caso del Mutex vince chi mette il lock e solo lui può toglierlo. Nel caso del semaforo c'è un meccanismo di priorità logica che permette a qualcun altro di prendere in mano la risorsa.

Python non permette di fare thread! Python è un linguaggio pensato per essere semplice, nel senso che impedisce di fare troppe cavolate.
Python è un linguaggio fortemente tipizzato. Non dichiaro mai le variabili, ma dopo che faccio `a = 1`, da quel punto la variabile è un intero e non posso cambiarlo, non posso successivamente scrivere `a = 1.5`.
GIL = Global Interpreter Lock. Si possono definire i thread, ma fisicamente non vengono runnati insieme, bensì in modo seriale. Allora perché farlo?
I thread sono utili quando è necessario fare I/O.
Se ho un thread che deve accedere a un file, python me lo permette.

Se voglio fare roba concorrenziale di **calcolo** contemporaneamente? Dobbiamo utilizzare i *processi*: istanze di programmi.

Posso dire che tre funzioni all'interno di un programma vengano fatte in processi differenti, che effettivamente runneranno in parallelo. Questo ha lo svantaggio che i processi abbiano memorie differenti, quindi devo trovare un modo per farle comunicare. Ho però il vantaggio di poter uccidere (kill) un singolo processo.

C'è un altro modo per fregare GIL, ovvero non usare python. Ad esempio quando usiamo alcune librerie, wrappate in python, ma scritte in C. E quelle librerie al loro interno usano i thread!

controllare di avere i moduli "multiprocessing" e "threading"

Process: pros and cons

pros:
-A process is an instance of a program, managed by operating system (memory space allocated by the kernel).
-Two processes can executecode simultaneously in thesame python program
-Separated memory space
-Takes advantage of multiple cores and CPUs
-Child processes are killable
-Avoid GIL limitations
cons:
-Relatively high overhead
-Open and close processes takes more time
-Sharing information between processes is very slow
-Model not adaptable to parallelism

Threads: pros and cons

pros:
-Processes produce threads (sub-processes) to handle sub-tasks (threads live inside the process and share the same memory space)

-Can use shared memory
-Threads communication
-Lightweight
-Very small overhead
-Great option for I/O bound application
cons:
-Subject to GIL (although there are workarounds)
-Not killable
-Potential of race condition
-Same memory space

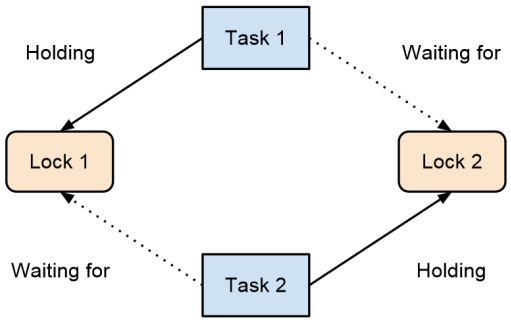
When to use threads vs processes?

Processes speed up Python operations that are CPU intensive because they benefit from multiple cores and avoid the GIL.
Threads are best for IO tasks or tasks involving external systems because threads can combine their work more efficiently. Processes need to pickle their results to combine them which takes time.

Threads provide no benefit in python for CPU intensive tasks because of the GIL.

Things to be afraid of! (not only in python...)

Starvation: a task is costantly denied necessary resource. The task can never finish (starves).
Deadlock: Usually a deadlock occurs when two or more tasks wait cyclically for each other.



Lun 24 ott - Lezione 10

The multiprocessing module

HelloWorld

Create a process to run the function f()

```
from multiprocessing import Process

def f(name):
    print('Hello '+name)

#MAIN
if __name__=="__main__":
    p = Process(target=f, args=('World',))
    p.start()
    p.join()
```

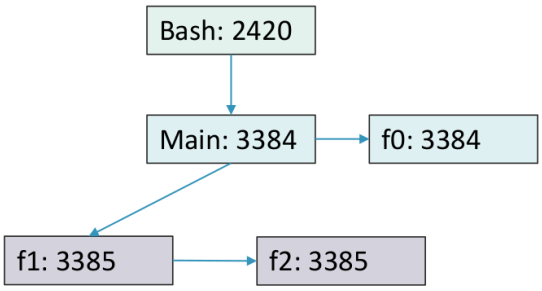
Trasformeremo quello che fa la funzione in un processo.
Una volta definito il processo, lo dobbiamo fare partire usando il metodo `p.start`. L'esecuzione di un thread può avvenire in modo sincrono e asincrono. Tipicamente avviene in modo asincrono: quando l'interprete trova `p.start` avvia il processo. Il processo parte; Il controllo del flusso va direttaente alla riga successiva, indipendentemente dl fatto che il processo sia terminato. Questo succede a meno che non utilizziamo `p.join`. In tal caso il processo avviene in modo sincrono: finché non è finito il processo, si aspetta.

FatherAndSons

Generate a tree of processes

```
from multiprocessing import Process
import os
def f0(name):
    print()
    print("-----> function "+name)
    print ("I am still the main process with ID "
           +str(os.getpid())+" my father is ID:"+str(os.getppid()))

def f1(name):
    print()
    print("-----> function "+name)
    print ("I am the first sub-process with ID "
           +str(os.getpid())+" my father is ID:"+str(os.getppid()))
    f2('two')


def f2(name):
    print()
    print("-----> function "+name)
    print ("I am still the first sub-process with ID "
           +str(os.getpid())+" my father is ID:"+str(os.getppid()))
    print("This is the end!")

#MAIN
if __name__=="__main__":
    print ("I am the main process with ID: "+str(os.getpid()))
    f0('zero')
    p = Process(target=f1, args=('one',))
    p.start()
    p.join()
```

Ho un processo main in cui viene chiamata la funzione f0. Il main genera un processo, definito dalla funzione f1, nel quale viene chiamata f2.



Su linux, se sulla linea di comando scriviamo `ps`, mi dice quali processi sono in esecuzione.

Use the Queue to get the result from multiple processes

```
import multiprocessing as mp

# define a example function
def Hello(pos,name):
    msg="Hello "+name
    output.put((pos, msg))

if __name__=="__main__":
    # Define an output queue
    output = mp.Queue()

    # Setup a list of processes that we want to run
    processes = [mp.Process(target=Hello, args=(x, "Gianluca")) for x in range(4)]

    # Run processes
    for p in processes:
        p.start()

    # Exit the completed processes
    for p in processes:
        p.join()

    # Get process results from the output queue
    results = [output.get() for p in processes]

    print(results)
```

La queue è una scatola in cui mettiamo dentro il risultato dei vari processi, per poi aprirla nel main.
nota: non possiamo assumere l'ordine delle cose che facciamo. Lo scheduler decide quando far partire i processi, che potrebbero finire in un ordine diverso da quello atteso.

How to distribute work to workers (aka cpu cores)

Use the Pool class.
Try `Pool.map`
Try `Pool.map_async`
See also `Pool.apply` e `Pool.apply_async`

```
def cube(x):
    print (str(os.getpid())+" "+str(os.getppid()))
```

```
    return x**3
#MAIN
if __name__=="__main__":
    pool = mp.Pool(processes=4)
    results = pool.map(cube,range(1,7))
    print(results)
```

```
#MAIN
if __name__=="__main__":
    pool = mp.Pool(processes=4)
    results = pool.map_async(cube,range(1,7))
    print(results.get())
```

nota i processi avviati con pool.map sono di per sé sincroni (e partono subito), perciò non serve usare `join` (e `start`).

Another example with `pool.map` and `pool.map_async`

Notice the time measurement

```
import multiprocessing as mp
import time
import os
def doingstuffs(x):
    print ("Process: "+str(x)+" "+str(os.getpid()))
    time.sleep(1)
if __name__=="__main__":
    start=time.time()
    pool = mp.Pool(processes=4)
    results = pool.map(doingstuffs,range(1,10))
    end=time.time()
    print("elapsed time: "+str(end-start))
```

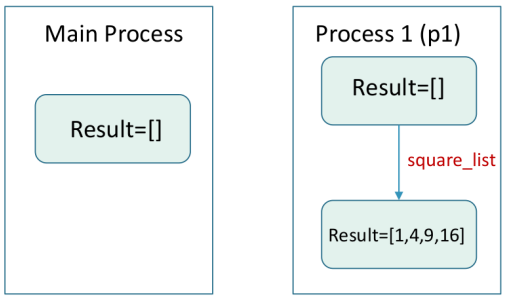
```
results = pool.map_async(doingstuffs,range(1,10))
#...
print(results.get())
```

Communication between processes

Un modo per (*illuderci di*) passare informazione da un processo al'altro è utilizzare le variabili globali.

```
import multiprocessing

# empty list with global scope
result = []

def square_list(mylist):
        global result
        for num in mylist:
                result.append(num * num)
        print("Result(in process p1): "+str(result))

#MAIN
if __name__=="__main__":
    # input list
    mylist = [1,2,3,4]
    # creating new process
```

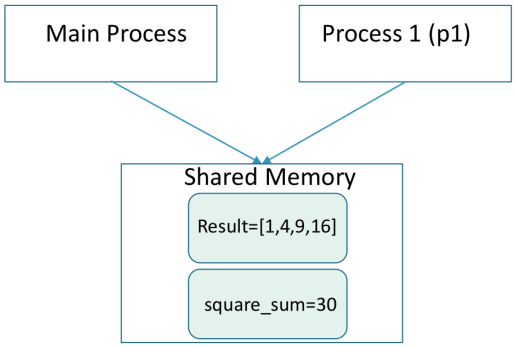
```
    p1 = multiprocessing.Process(target=square_list, args=(mylist,))
    # starting process
    p1.start()
    # wait until process is finished
    p1.join()

    # print global result list
    print("Result(in main program): "+str(result))

```



Different memory spaces allocated for each process. Try to print result in both processes.

Comm. between processes: shared memory

Normalmente abbiamo visto che le memorie sono separate. É possibile definire una zona di memoria (*shared memory*) comune ad entrambi i processi.



Shared memory: multiprocessing module provides Array and Value objects to share data between processes.
`Array`: array allocated from shared memory.
`Value`: object allocated from shared memory.

```
import multiprocessing

def square_list(mylist, result, square_sum):
    for idx, num in enumerate(mylist):
        result[idx] = num * num
    # square_sum value
    square_sum.value = sum(result)
    # print result Array
    print("Result(in process p1): "+str(result[:]))
    # print square_sum Value
    print("Sum of squares(in process p1): "+str(square_sum.value))

if __name__=="__main__":
    # input list
    mylist = [1,2,3,4]
```

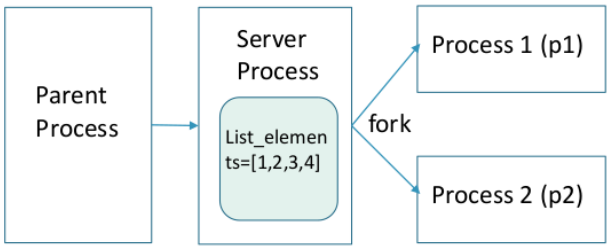
```
    # creating Array of int data type with space for 4 integers
    result = multiprocessing.Array('i', 4)
    # creating Value of int data type
    square_sum = multiprocessing.Value('i')
    # creating new process
    p1 = multiprocessing.Process(target=square_list, args=(mylist, result, square_sum))

    # starting process
    p1.start()
    # wait until process is finished
    p1.join()

    # print result array
    print("Result(in main program): "+str(result[:]))
    # print square_sum Value
    print("Sum of squares(in main program): "+str(square_sum.value))

```

Nella shared memory non posso mettere oggetti complicati come i dizionari.

Comm. between processes: server process

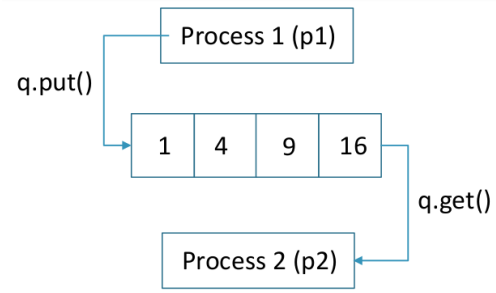
Server process : Whenever a python program starts, a server process is also started. From there on, whenever a new process is needed, the parent process connects to the server and requests it to fork a new process. A server process can hold Python objects and allows other processes to manipulate them. multiprocessing module provides a Manager class which controls a server process. Hence, managers provide a way to create data which can be shared between different processes. Server process allows to share any type of object (dict, lists,...). It is also possible to connect a server process to the network

```
import multiprocessing

def add_element(record,records):
    records.append(record)
    print("New element added to records list")

def sum_elements(records):
    summ=sum(records)
    print("New sum is: "+str(summ))

#MAIN
with multiprocessing.Manager() as manager:
    list_elements=[1,2,3,4]
    records=manager.list(list_elements)
    new_element=5

    print("Old sum is: "+str(sum(list_elements)))
    #creating new processes
    p1 = multiprocessing.Process(target=add_element, args=(new_element,records))
    p2= multiprocessing.Process(target=sum_elements, args=(records,))

    #running process p1 to insert new element
    p1.start()
    p1.join

    #running process p2 to sum list elements
    p2.start()
    p2.join()

```



Comm. between processes: queue

Queue : A simple way to communicate between process with multiprocessing is to use a Queue to pass messages back and forth.
Any Python object can pass through a Queue.

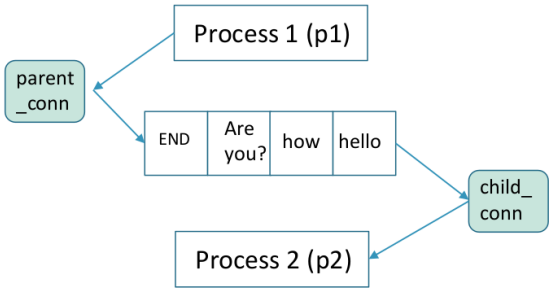



```python
import multiprocessing

def square_list(mylist, q):
    # append squares of mylist to queue
    for num in mylist:
        q.put(num * num)

def print_queue(q):
    print("Queue elements:")
    while not q.empty():
        print(q.get())
    print("Queue is now empty!")

#MAIN
if __name__=="__main__":
    # input list
    mylist = [1,2,3,4]
    # creating multiprocessing Queue
    q = multiprocessing.Queue()

    # creating new processes
    p1 = multiprocessing.Process(target=square_list, args=(mylist, q))
    p2 = multiprocessing.Process(target=print_queue, args=(q,))

    # running process p1 to square list
    p1.start()
    p1.join()
    # running process p2 to get queue elements
    p2.start()
    p2.join()
```

nota: quando estraggo un elemento dalla coda, lo rimuovo da essa.

Comm. between process: pipe

In linea di principio, la coda permette di avere più *endpoint*: non necessariamente entra da un lato ed esce da un altro. Invece la pipe è così: la dobbiamo immaginare proprio come un tubo.



Se ho soltanto due processi: uno che scrive e uno che legge, allora è più conveniente usare le pipe perché sono più veloci.

Pipes : A pipe can have only two endpoints. Hence, it is preferred over queue when only two-way communication is required. Queue is slower (it's built on top of pipe).

multiprocessing module provides `Pipe()` function which returns a pair of connection objects connected by a pipe. The two connection objects returned by `Pipe()` represent the two ends of the pipe. Each connection object has `send()` and `recv()` methods (among others).

Synchronization between processes

Process synchronization is defined as a mechanism which ensures that two or more concurrent processes do not simultaneously execute some particular program segment known as critical section. A race condition occurs when two or more processes can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

```
1  import multiprocessing
2
3  def withdraw(balance):
4          for x in range(10000):
5                  balance.value = balance.value-1
6  def deposit(balance):
7          for x in range(10000):
8                  balance.value = balance.value + 1
9
10 def perform_transactions():
11         # initial balance (in shared memory)
12         balance = multiprocessing.Value('i', 100)
13         # creating new processes
14         p1 = multiprocessing.Process(target=withdraw, args=(balance,))
15         p2 = multiprocessing.Process(target=deposit, args=(balance,))
16         # starting processes
17         p1.start()
18         p2.start()
19         # wait until processes are finished
20         p1.join()
21         p2.join()
22         # print final balance
23         print("Final balance = {}".format(balance.value))
24
25 #MAIN
26 for x in range(10):
27     # perform same transaction process 10 times
28     perform_transactions()
```

Se permettiamo a due processi di scrivere contemporaneamente sulla stessa locazione di memoria succede un casino!

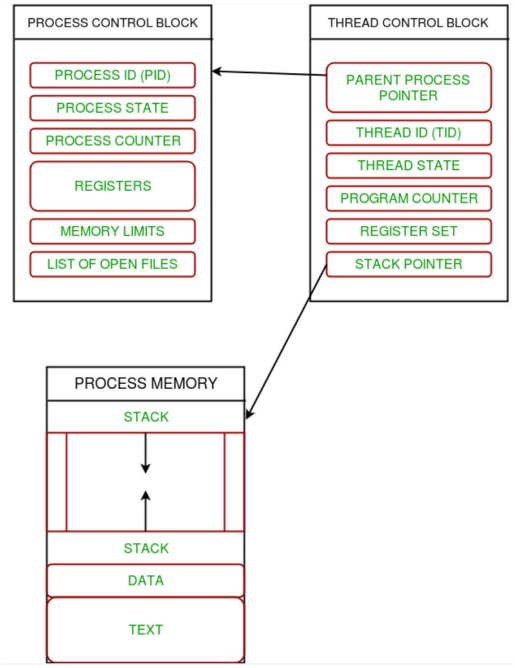
multiprocessing module provides a Lock class to deal with the race conditions. Lock is implemented using a Semaphore object provided by the Operating System. A semaphore is a synchronization object that controls access by multiple processes to a common resource in a parallel programming environment. It is simply a value in a designated place in operating system (or kernel) storage that each process can check and then change. Depending on the value that is found, the process can use the resource or will find that it is already in use and must wait for some period before trying again.

```
import multiprocessing

# function to withdraw from account
def withdraw(balance, lock):
    for x in range(10000):
        lock.acquire()
        balance.value = balance.value - 1
        lock.release()

# function to deposit to account
def deposit(balance, lock):
    for x in range(10000):
        lock.acquire()
        balance.value = balance.value + 1
        lock.release()

def perform_transactions():
        # initial balance (in shared memory)
        balance = multiprocessing.Value('i', 100)
        # creating a lock object
        lock = multiprocessing.Lock()

        # creating new processes
        p1 = multiprocessing.Process(target=withdraw, args=(balance,lock))
        p2 = multiprocessing.Process(target=deposit, args=(balance,lock))
        # starting processes
        p1.start()
        p2.start()
        # wait until processes are finished
        p1.join()
        p2.join()

        # print final balance
        print("Final balance = "+str(balance.value))

#MAIN
if __name__=="__main__":
    for x in range(10):
        # perform same transaction process 10 times
        perform_transactions()
```

Il lock si utilizza ogni volta che si vuole impedire che la stessa risorsa venga usata due volte.

Threading

A thread is an entity within a process that can be scheduled for execution. Also, it is the smallest unit of processing that can be performed in an OS (Operating System).
In simple words, a thread is a sequence of such instructions within a program that can be executed independently of other code. For simplicity, you can assume that a thread is simply a subset of a process! Multiple threads can exist within one process where:
-Each thread contains its own register set and local variables (stored in stack).
-All thread of a process share global variables (stored in heap) and the program code.



Threading module

The threads aren't different processes. Due to GIL the parallelism is only «Logic».

```
1  import threading
2  import os
3
4  def task1():
5          print("Task 1 assigned to thread: "+threading.current_thread().name)
6          print("ID of process running task 1: "+str(os.getpid()))
7  def task2():
8          print("Task 2 assigned to thread: "+threading.current_thread().name)
9          print("ID of process running task 2: "+str(os.getpid()))
10 #MAIN
11 if __name__=="__main__":
12         # print ID of current process
13         print("ID of process running main program: "+str(os.getpid()))
14         # print name of main thread
15         print("Main thread name: "+threading.main_thread().name)
16
17         # creating threads
18         t1 = threading.Thread(target=task1, name='t1')
19         t2 = threading.Thread(target=task2, name='t2')
20         # starting threads
21         t1.start()
22         t2.start()
23         # wait until all threads finish
24         t1.join()
25         t2.join()
```

Threads synchronization

```
1  import threading
2
3  # global variable x
4  x = 0
5
6  def increment():
```

```
    global x
    x += 1

def thread_task():
    for _ in range(100000):
        increment()

def main_task():
    global x
    # setting global variable x as 0
    x = 0
    # creating threads
    t1 = threading.Thread(target=thread_task)
    t2 = threading.Thread(target=thread_task)

    # start threads
    t1.start()
    t2.start()
    # wait until threads finish their job
    t1.join()
    t2.join()

#MAIN
for i in range(10):
    main_task()
    print("Iteration {0}: x = {1}".format(i,x))
```

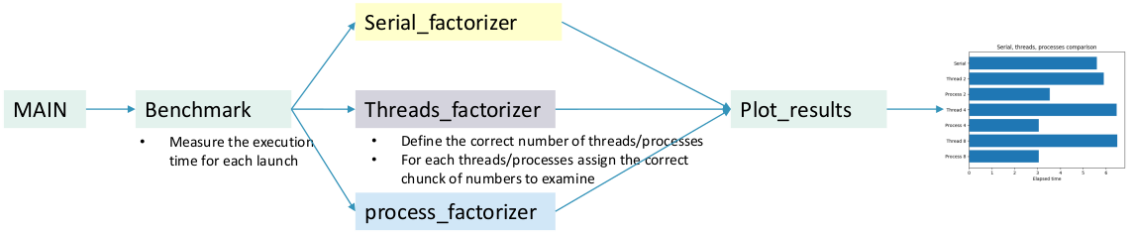
```
import threading

# global variable x
x = 0

def increment():
    global x
    x += 1

def thread_task(lock):
    for _ in range(100000):
        lock.acquire()
        increment()
        lock.release()

def main_task():
    global x
    # setting global variable x as 0
    x = 0
    # creating a lock
    lock = threading.Lock()

    # creating threads
    t1 = threading.Thread(target=thread_task, args=(lock,))
    t2 = threading.Thread(target=thread_task, args=(lock,))

    # start threads
    t1.start()
    t2.start()
    # wait until threads finish their job
    t1.join()
    t2.join()

#MAIN
for i in range(10):
```

```
    main_task()
    print("Iteration {0}: x = {1}".format(i,x))
```

Comparison between Threads and Processes

Write a code to factorize a list of numbers: the 300 odd numbers from 1000000000001 and 1000000000597.

Try to benchmark the time needed to factorize this list by using:
-Serial code
-2,4,8 Threads
-2,4,8 Processes
Produce a plot with the results

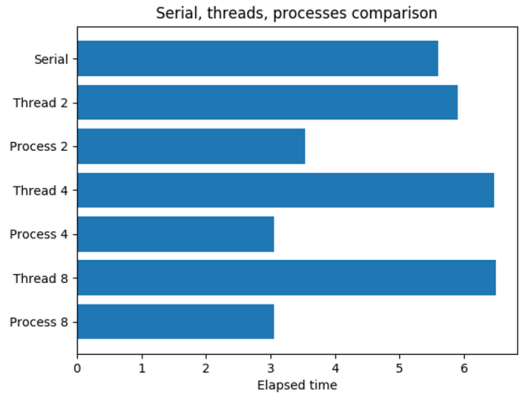



```
import math
import multiprocessing
import random
import threading
import time
import matplotlib.pyplot as plt
import numpy


class Timer(object):
    def __init__(self, name=None):
        self.name = name
        self.timee=0

    def __enter__(self):
        self.tstart = time.time()

    def __exit__(self, type, value, traceback):
        if self.name:
            print('[%s]' % self.name, end=' ')
        self.timee=(time.time() - self.tstart)
        print('Elapsed: %s' % (time.time() - self.tstart))
        self.output()

    def output(self):
        return self.timee


def factorize_naive(n):
    """ A naive factorization method. Take integer 'n', return list of
        factors.
    """
    if n < 2:
        return []
    factors = []
    p = 2

```

```
    while True:
        if n == 1:
            return factors
        r = n % p
        if r == 0:
            factors.append(p)
            n = n // p
        elif p * p >= n:
            factors.append(n)
            return factors
        elif p > 2:
            # Advance in steps of 2 over odd numbers
            p += 2
        else:
            # If p == 2, get to 3
            p += 1
    assert False, "unreachable"


# Each "factorizer" function returns a dict mapping num -> factors
def serial_factorizer(nums):
    return {n: factorize_naive(n) for n in nums}

def threaded_factorizer(nums, nthreads):
    def worker(nums, outdict):
        """ The worker function, invoked in a thread. 'nums' is a
            list of numbers to factor. The results are placed in
            outdict.
        """
        for n in nums:
            outdict[n] = factorize_naive(n)

    # Each thread will get 'chunksize' nums and its own output dict
    chunksize = int(math.ceil(len(nums) / float(nthreads)))
    threads = []
    outs = [{} for i in range(nthreads)]

    for i in range(nthreads):
        # Create each thread, passing it its chunk of numbers to factor
        # and output dict.
        t = threading.Thread(
                target=worker,
                args=(nums[chunksize * i:chunksize * (i + 1)],
                      outs[i]))
        threads.append(t)
        t.start()

    # Wait for all threads to finish
    for t in threads:
        t.join()

    # Merge all partial output dicts into a single dict and return it
    return {k: v for out_d in outs for k, v in out_d.items()}

def mp_worker(nums, out_q):
    """ The worker function, invoked in a process. 'nums' is a
        list of numbers to factor. The results are placed in
        a dictionary that's pushed to a queue.
    """
    outdict = {}
    for n in nums:
        outdict[n] = factorize_naive(n)
    out_q.put(outdict)
```

```
101
102
103 def mp_factorizer(nums, nprocs):
104     # Each process will get 'chunksize' nums and a queue to put his out
105     # dict into
106     out_q = multiprocessing.Queue()
107     chunksize = int(math.ceil(len(nums) / float(nprocs)))
108     procs = []
109
110     for i in range(nprocs):
111         p = multiprocessing.Process(
112                 target=mp_worker,
113                 args=(nums[chunksize * i:chunksize * (i + 1)],
114                       out_q))
115         procs.append(p)
116         p.start()
117
118     # Collect all results into a single result dict. We know how many dicts
119     # with results to expect.
120     resultdict = {}
121     for i in range(nprocs):
122         resultdict.update(out_q.get())
123
124     # Wait for all worker processes to finish
125     for p in procs:
126         p.join()
127
128     return resultdict
129
130 def plot_results(elapsed):
131     plt.rcdefaults()
132     fig, ax = plt.subplots()
133     laby = ('Serial','Thread 2','Process 2','Thread 4','Process 4','Thread 8','Process 8')
134     y_pos = numpy.arange(len(laby))
135     ax.barh(y_pos, elapsed, align='center')
136     ax.set_yticks(y_pos)
137     ax.set_yticklabels(laby)
138     ax.invert_yaxis()  # labels read top-to-bottom
139     ax.set_xlabel('Elapsed time')
140     ax.set_title('Serial, threads, processes comparison')
141     plt.show()
142     wait()
143
144 def benchmark(nums):
145     print('Running benchmark...')
146     elapsed_times=[]
147
148     tserial=Timer('serial')
149     with tserial as qq:
150         s_d = serial_factorizer(nums)
151     elapsed_times.append(tserial.output())
152
153     for numparallel in [2, 4, 8]:
154         tthread=Timer('threaded %s' % numparallel)
155         with tthread as qq:
156             t_d = threaded_factorizer(nums, numparallel)
157         elapsed_times.append(tthread.output())
158         tmpar=Timer('mp %s' % numparallel)
159         with tmpar as qq:
160             m_d = mp_factorizer(nums, numparallel)
161         elapsed_times.append(tmpar.output())
162
163     print (elapsed_times)
```

```
164     plot_results(elapsed_times)
165
166
167 #MAIN
168 N = 299
169
170 nums = [999999999999]
171 for i in range(N):
172     nums.append(nums[-1] + 2)
173 print(nums)
174 benchmark(nums)
```



Why should I use threads?

GIL is bypassed in two cases:
-running programs in external C code (ex: numpy)
-in case of I/O operation: Python release the lock waiting for I/O

A tipical application is the use of the network. Writing to a disk, display an image to the screen, print on a printer,...

```
1  import requests
2  import threading as thr
3  from time import perf_counter
4
5  buffer_size=1024
6  #define a function to manage the download
7  def download(url):
8          response = requests.get(url, stream=True)
9          filename = url.split("/")[-1]
10         with open(filename,"wb") as f:
11                 for data in response.iter_content(buffer_size):
12                         f.write(data)
13
14 #MAIN
15 if __name__ == "__main__":
16         urls= [
17                 "http://cds.cern.ch/record/2690508/files/201909-262_01.jpg",
18                 "http://cds.cern.ch/record/2274473/files/05-07-2017_Calorimeters.jpg",
19                 "http://cds.cern.ch/record/2274473/files/08-07-2017_Spectrometer_magnet.jpg",
20                 "http://cds.cern.ch/record/2127067/files/_MG_3944.jpg",
21                 "http://cds.cern.ch/record/2274473/files/08-07-2017_Electronics.jpg",
22         ]
23
24         t = perf_counter()
```

```
#sequential download
        for url in urls:
                download(url)
        print("Time: "+str(perf_counter()-t))
```

Versione parallela: faccio 5 thread che scaricano contemporaneamente 5 immagini:

```
import threading as thr
import requests
import os
from time import perf_counter

buffer_size=1024

#define a function to manage the download
def download(url):
        response = requests.get(url, stream=True)
        filename = url.split("/")[-1]
        with open(filename,"wb") as f:
                for data in response.iter_content(buffer_size):
                        f.write(data)


#MAIN
if __name__ == "__main__":
        urls= [
                "http://cds.cern.ch/record/2690508/files/201909-262_01.jpg",
                "http://cds.cern.ch/record/2274473/files/05-07-2017_Calorimeters.jpg",
                "http://cds.cern.ch/record/2274473/files/08-07-2017_Spectrometer_magnet.jpg",
                "http://cds.cern.ch/record/2127067/files/_MG_3944.jpg",
                "http://cds.cern.ch/record/2274473/files/08-07-2017_Electronics.jpg",
        ]

#define 5 threads
        threads = [thr.Thread(target=download, args=(urls[x],)) for x in range(4)]

        t = perf_counter()

#start threads
        for thread in threads:
                thread.start()

#join threads
        for thread in threads:
                thread.join()

        print("Time: "+str(perf_counter()-t))
```

Performaces depend on network speed. Overheads for thread start and lock release.

Process vs Threads

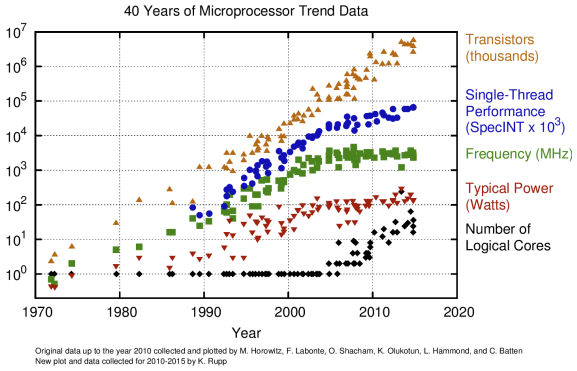
Process	Thread
Separate memory	Shared memory
More memory	Less memory
Killable children (but can become zombies)	No zombies
More overhead	Less Overhead
Slower creation and destruction	Faster creation and destruction
Easier to code and debug	Harder to code and debug
No GIL: yes for CPU-bound problems	GIL: No for CPU-bound problems (ok for I/O)

Gio 27 ottobre - Lezione 11

Introduction to GPU computing (1)

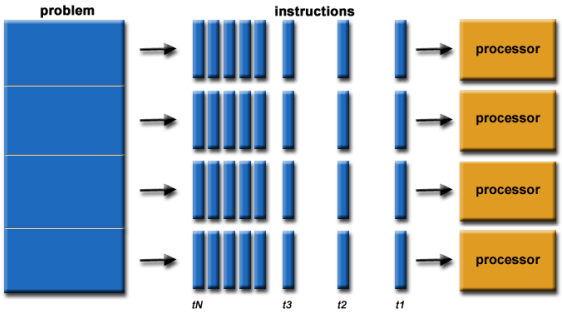
Moore's Law

Moore's law: "The performance of microprocessors and the number of their transistors will double every 18 months".
The increasing of performance is related to the clock.
Faster clock means higher dissipation → power wall



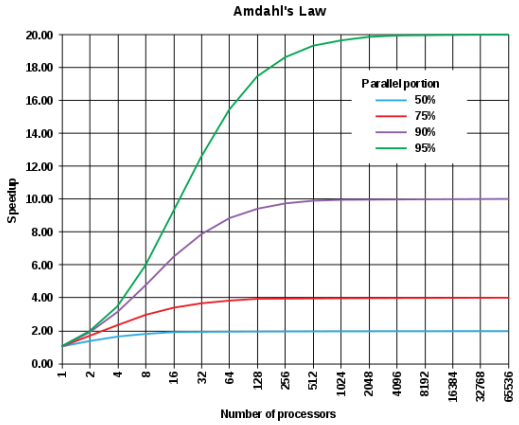
Parallel programming

Parallel computing is no longer something for SuperComputers. All the processors nowadays are multicores.
The use of parallel architectures is mainly due to the physical constraints to frequency scaling.



Limits of parallel programming

Several problems can be split in smaller problems to be solved concurrently. In any case the maximum speedup is not linear , but it depends on the serial part of the code (Amdahls's law)
The situation can improve if the amount of parallelizable part depends on the resources(Gustafson's Law)



$$S_{latency} = \frac{1}{1 - p + \frac{p}{s}}$$

$$S_{latency} = 1 - p + sp$$

What are GPUs?

The GPUs are processors dedicated to parallel programming for graphical application. Rendering, Image transformation, ray tracing, etc. are typical application where parallelization can helps a lot.

Standard GPU pipeline

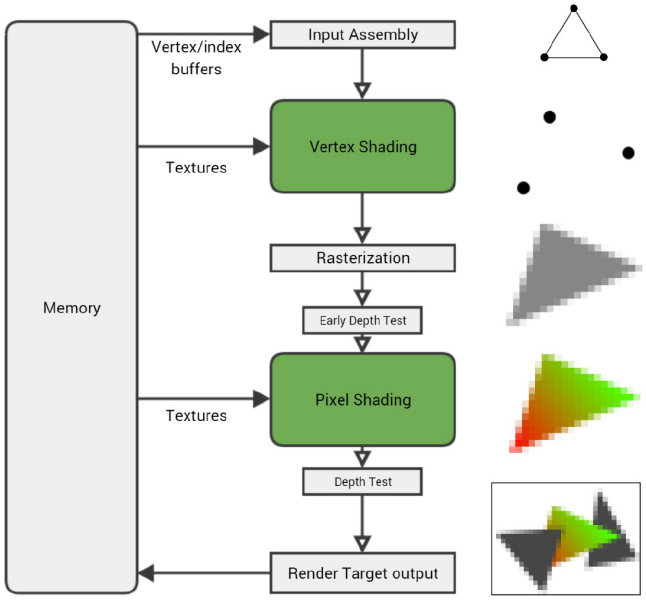
Vertex/index buffers:
Description of image with vertices and their connection to triangles

Vertex shading
For every vertex: calculate position on screen based on original position and camera view point

Rasterization
Get per-pixel color values

Pixel shading
For every pixel: get color based on texture properties (material, light, ...)

Rendering
Write output to render target



http://fragmentbuffer.com/gpu-performance-for-game-artists/

Ogni triangolino è indipendente dall'altro. Possiamo agire contemporaneamente su questi triangolini in maniera parallela.

Standard GPU requirements

Graphics pipeline: huge amount of arithmetic on independent data:
-Transforming positions
-Generating pixel colors

-Applying material properties and light situation to every pixel
Hardware needs
-Access memory simultaneously and contiguously
-Bandwidth more important than latency
-Floating point and fixed-function logic

What are the GPUs?

The technical definition of a GPU is "a single-chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second.“
The possibility to use the GPU for generic computing (GPGPU) has been introduced by NVIDIA in 2007 (CUDA)
In 2008 OpenCL: consortium of different firms to introduce a multi-platform language for many-cores computing.

Why the GPUs?

-GPU is a way to cheat the Moore's law
Implementano la possibilità di eseguire operazioni ad una velocità superiore a quella del clock.
-SIMD/SIMT parallel architecture
-The PC no longer get faster, just wider.
-Very high computing power for «vectorizable» problems
-Impressive derivative almost a factor of 2 in each generation
-Continuous development
-Easy to have a desktop PC with teraflops of computing power, with thousand of cores.
-Several applications in HPC, simulation, scientific computing...

Vogliamo imparare a sfruttare una tecnologia utilizzata sul mercato (che migliora ogni anno), per i nostri scopi di calcolo scientifico.

A lot of cores...

	"Fermi"	"Fermi"	"Kepler"	"Kepler"	"Maxwell"	"Pascal"
Tesla GPU	**GF100**	**GF104**	**GK104**	**GK110**	**GM200**	**GP100**
Compute Capability	2.0	2.1	3.0	3.5	5.3	6.0
Streaming Multiprocessors (SMs)	16	16	8	15	24	56
FP32 CUDA Cores / SM	32	32	192	192	128	64
FP32 CUDA Cores	512	512	1536	2880	3072	3584
FP64 Units	-	-	512	960	96	1792
Threads / Warp	32	32	32	32	32	32
Max Warps / Multiprocessor	48	48	64	64	64	64
Max Threads / Multiprocessor	1536	1536	2048	2048	2048	2048
Max Thread Blocks / Multiprocessor	8	8	16	16	32	32
32-bit Registers / Multiprocessor	32768	32768	65536	65536	65536	65536
Max Registers / Thread	63	63	63	255	255	255
Max Threads / Thread Block	1024	1024	1024	1024	1024	1024
Shared Memory Size Configurations	16 KB	16 KB	16 KB	16 KB	96 KB	64 KB
	48 KB	48 KB	32 KB	32 KB		
			48 KB	48 KB		
Hyper-Q	No	No	No	Yes	Yes	Yes
Dynamic Parallelism	No	No	No	Yes	Yes	Yes
Unified Memory	No	No	No	No	No	Yes
Pre-Emption	No	No	No	No	No	Yes

GPU Features	GTX 1080Ti	RTX 2080 Ti	Quadro P6000	Quadro RTX 6000
Architecture	Pascal	Turing	Pascal	Turing
GPCs	6	6	6	6
TPCs	28	34	30	36
SMs	28	68	30	72
CUDA Cores / SM	128	64	128	64
CUDA Cores / GPU	3584	4352	3840	4608
Tensor Cores / SM	NA	8	NA	8
Tensor Cores / GPU	NA	544	NA	576
RT Cores	NA	68	NA	72
GPU Base Clock MHz (Reference / Founders Edition)	1480 / 1480	1350 / 1350	1506	1455

Metrics

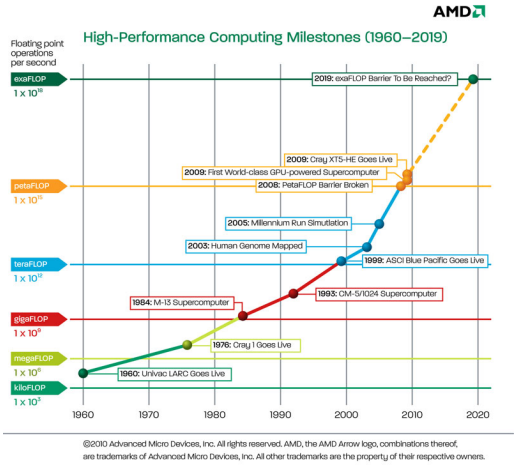
FLOPS (Floating Point operation per second):
It is a measurement of the computing power of a processor. Theoretically is defined as:

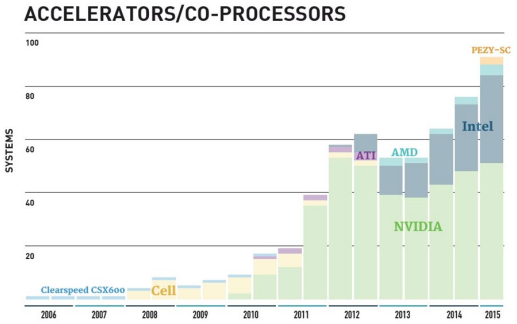
$$FLOPS = clock * cores * Operation/cycle$$

Actually this formula doesn't take into account several things
The real extimation is made experimentally by using standard packages (LINPACK, LAPACK)
The FLOPS is only a first indication of the computing power.
Other metrics has been invented to avoid the limitations of the FLOPS.

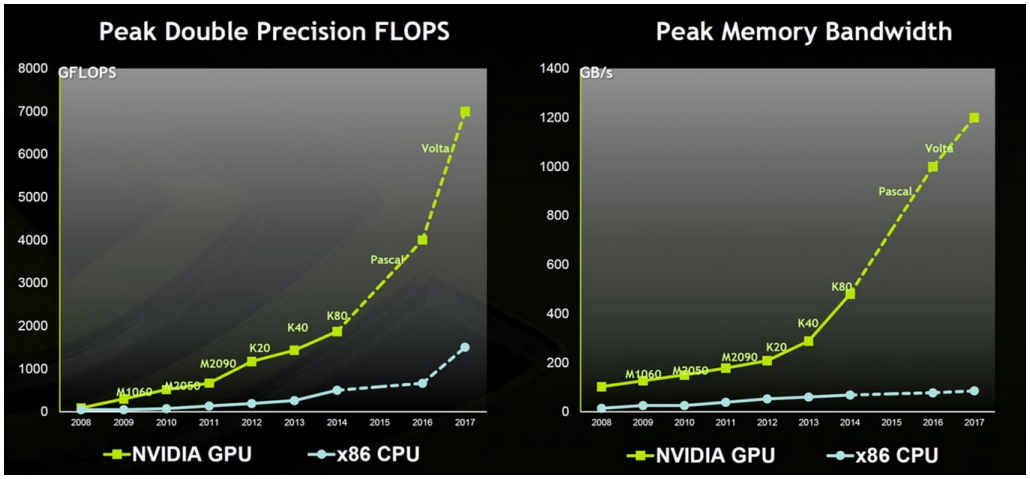
SPECint and SPECf

They are suites of 12 benchmarks of different type (for integer and floating point). The extimation is relative to a particular machine.





Computing power comparison



CPU

-**Multilevel and Large Caches**: Convert long latency memory access to -short latency cache latency.
-**Branch prediction:** To reduce latency in branching -Instruction level parallelism (ILP)
-Powerful ALU: Reduced operation latency
-Memory management
-Large control part

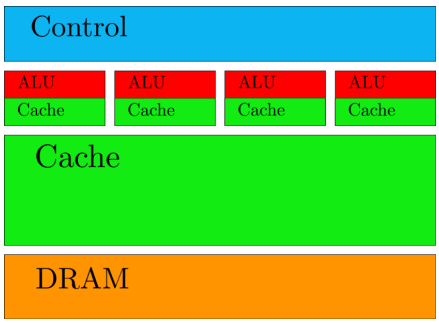


Figure 2.11: CPU: latency oriented design

GPU

-SIMT/SIMD (Single instruction Multiple Thread/Data) architecture
-SMX (Streaming Multi Processors) to execute kernels
-Thread level parallelism: Massive threading to hide the latency
-Limited caching: To boost memory throughput
-Limited control
-No branch prediction, but branch predication

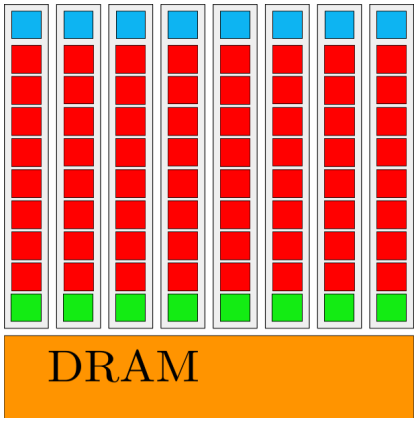


Figure 2.12: GPU: throughput oriented design

CPU vs GPU

- + Large main memory
- + Fast clock rate
- + Large caches
- + Branch prediction
- + Powerful ALU
- Relatively low memory bandwidth
- Cache misses costly
- Low performance per watt

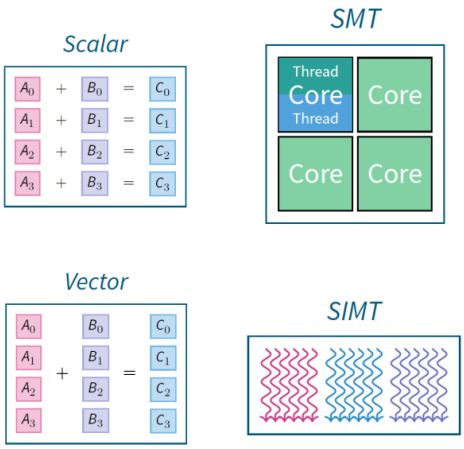
- + High bandwidth main memory
- + Latency tolerant (parallelism)
- + More compute resources
- + High performance per watt
- Limited memory capacity
- Low per-thread performance
- Extension card

	Intel Core E7-8890 v3	GeForce GTX 1080
Core count	18 cores / 36 threads	20 SMs / 2560 cores
Frequency	2.5 GHz	1.6 GHz
Peak Compute Performance	1.8 GFLOPs	8873 GFLOPs
Memory bandwidth	Max. 102 GB/s	320 GB/s
Memory capacity	Max. 1.54 TB	8 GB
Technology	22 nm	16 nm
Die size	662 mm^2	314 mm^2
Transistor count	5.6 billion	7.2 billion
Model	Minimize latency	Hide latency through parallelism

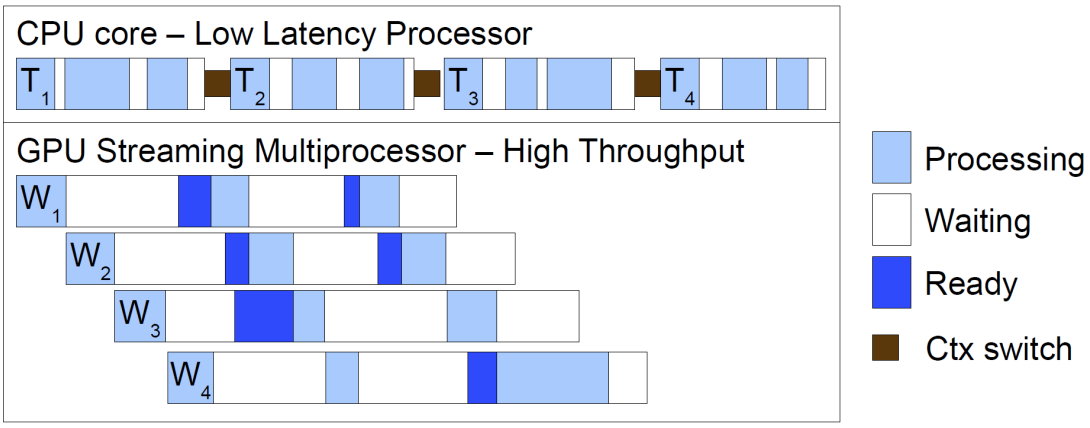
SIMT

Consideriamo un processore da 4000 core. La struttura SIMD prevederebbe che tutti i core facciano la stessa cosa nello stesso momento (ad es sommare due vettori di 4000 elementi).

- Standard CPU : Scalar processors
- SIMD CPU: vector processors
- Simultaneous threads in multicore processors
- SIMT (Single Instruction Multiple Threads)
  - CPU core GPU multiprocessor (SMX)
  - Working unit: a set of threads (32, a warp)
  - Fast switching of threads



CPU core vs GPU SMX

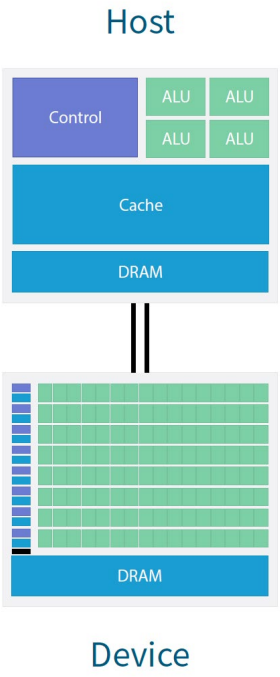


The latency in a SMX is hidden thanks to very deep pipelines.
The multithreading in a single CPU core is based on context switching.

GPU+CPU

The winning application uses both CPU and GPU:

- CPUs for sequential parts
  - can be 10X faster than GPU for sequential code
- GPUs for parallel part where throughput wins
  - can be 100X faster than CPU for parallel code
- The Host-Device connection is done with PCIe-gen3 (16 GB/s) or NVLINK (80 GB/s)
  - Relatively slow
  - Do as little as possible
- The bandwidth between GPU and video memory is HBM2 (720 GB/s in P100, 900 GB/s in V100)



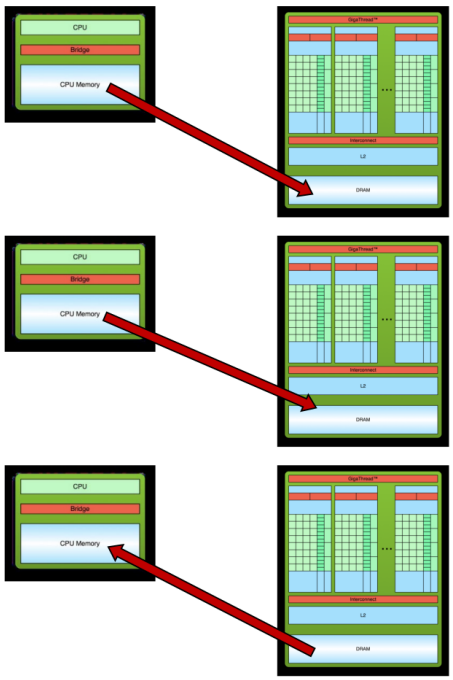
Summary:

- Superscalars, Pipelining and Vectorialization are methods to exploit some «parallelism» at the instruction and data level: ILP
  - Probably OOO (Out-of-order) execution should be included in this category
- The possible improvement thanks to ILP depends on problem and data structures
  - 1x-10x for Superscalars and Pipeline
  - 2x,4x,8x,16x for the vectorialization
- These methods show «saturation» because they are limited by the CPU resources available
  - Pentium 4: 30 pipeline stages (nowadays 10-15 maximum)
  - ARM A57 (Apple A7/A8): 9 ports/6 instructions superscalar
  - Intel Tiger Lake: vector of 512 bits for a subset of AVX512 instructions

... the point is: can CPU resources grow indefinitely?

Introduction to GPU computing (2)

CUDA model



CUDA is a set of C/C++ extensions to enable the GPGPU computing on NVIDIA GPUs. Dedicated APIs allow to control almost all the functions of the graphics processor.
Three steps:

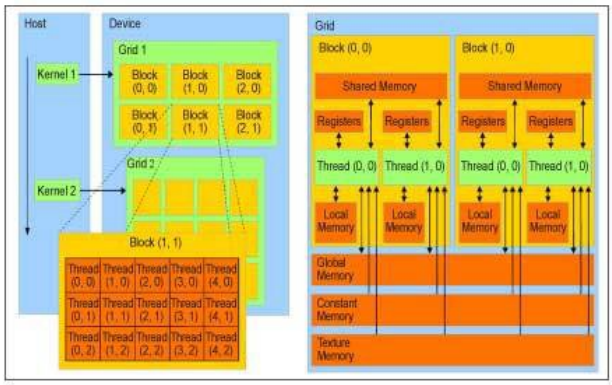
1. copy data from Host to Device
2. copy Kernel and execute
3. copy back results

Grid, blocks and threads

The computing resources are logically (and physically) grouped in a flexible parallel model of computation:

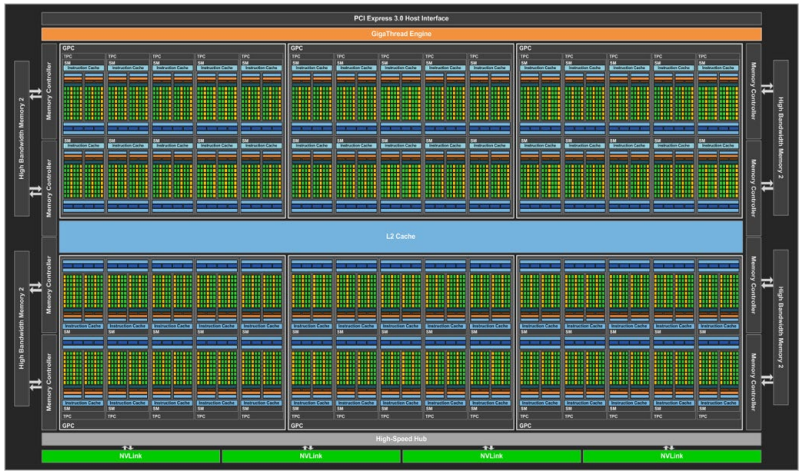
- 1D,2D and 3D grid
- With 1D, 2D and 3D blocks
- With 1D, 2D and 3D threads

Only threads can communicate and synchronize in a block.
Threads in different blocks do not interact, threads in same block execute same instruction at the same time.
The "shape" of the system is decided at kernel launch time.



GPU structure

I singoli thread runnano sui core. I thread sono raggruppate in blocchi. Un blocco è implementato da un multiprocessore.



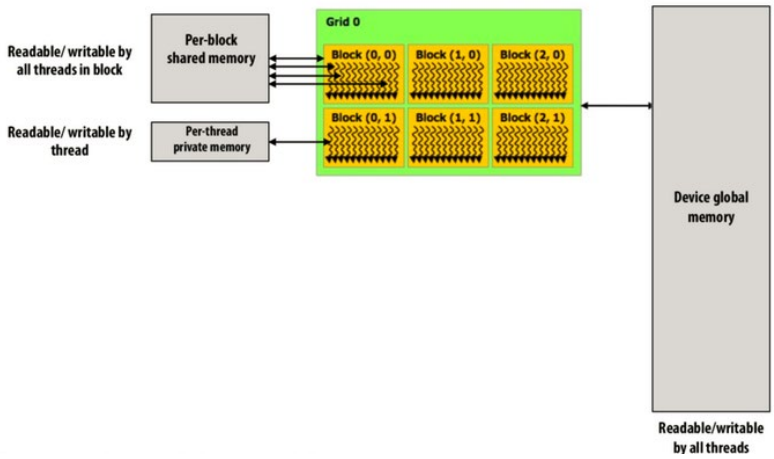
Multiprocessor

Anche se il numero di core è più piccolo del numero di thread, continua a poterli runnare, perché lo scheduler si prende il compito di fare eseguire i vari compiti in maniera parallela.

Memory

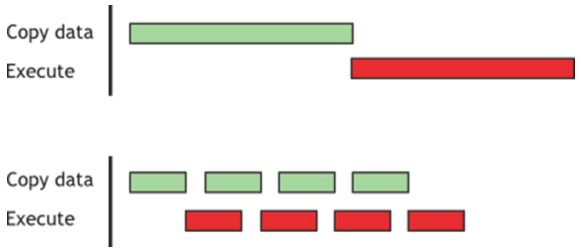
The memory hierarchy is fundamental in GPU programming. Most of the memory managing and data locality is left to the user.

- Unified Address Space
- Global Memory
  - On board, relatively slow, lifetime of the application, accessible from host and device
- Shared memory/registers
  - On Chip, very fast, lifetime of blocks/threads, accessible from kernel only



Asynchronicity

Problem: Memory transfer is comparably slow
Solution: Do something else in meantime (computation)!
Overlap tasks:
-Copy and compute engines run separately (streams) -GPU needs to be fed: Schedule many computations -CPU can do other work while GPU computes; synchronization



How to program GPU?

- CUDA is the "best" way to program NVIDIA GPU at "low level"
- If your code is almost CPU or if you need to accelerate dedicated functions, you could consider to use
  - Directives
    - OpenMP, OpenACC, ...
  - Libraries
    - Thrust, ArrayFire,...
- OpenCL is a framework equivalent to CUDA to program multiplatforms
  - GPU, CPU, DSP, FPGA,...
- C/C++ and Fortran are the "official" languages for CUDA
  - Python and other languages are supported through wrapping and libraries

Libraries: cuBLAS

GPU-parallel linear algebra routines (152 routines).
Single, double, complex data types
Possibility to use multiple GPUs
Example (among 152 routines): Saxpy: given two vectors x[10] and y[10] compute y[i]=a*x[i]+y[i]
`https://docs.nvidia.com/cuda/cublas/index.html`
`https://developer.nvidia.com/cublas`

```
int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y
cublasInit();
float * d_x, * d_y;
cudaMalloc((void **)&d_x, n * sizeof(x[0]);
cudaMalloc((void **)&d_y, n * sizeof(y[0]);
cublasSetVector(n, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(n, sizeof(y[0]), y, 1, d_y, 1);
cublasSaxpy(n, a, d_x, 1, d_y, 1);
cublasGetVector(n, sizeof(y[0]), d_y, 1, y, 1);
cublasShutdown();
```

Libraries: Thrust

-Template library
-Data parallel primitives (scan(), sort(), reduce(), ...)
-Comes when you install CUDA for free

```
int a = 42;
int n = 10;
thrust::host_vector<float> x(n), y(n);
// fill x, y
thrust::device_vector d_x = x, d_y = y;
using namespace thrust::placeholders;
thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_y.begin(), a * _1 + _2);
x = d_x;
```

Directives: OpenMP, OpenACC

The directive is the best transparent way to use GPU.
You must only «annotate» the part of the code you want to parallelize:

```
#pragma acc loop
for (int i = 0; i < 100; i++) {};
```

Hello world

Per compilare si usa il compilatore `nvcc`. Lo si installa installando CUDA dal sito NVIDIA.

```
nvcc -o HelloWorldGpu HelloWorldGpu.cu -arch=compute_30 -code=sm_30
```

Pro: Portability; easy to program
Cons: Not all the raw GPU power available; harder to debug; easy to program wrong

OpenACC is more focused on GPU, while OpenMP is for multi-computers (but still usable with GPU)

```
void saxpy_acc(int n, float a, float * x, float * y) {
        #pragma acc kernels
        for (int i = 0; i < n; i++) y[i] = a * x[i] + y[i];
}
...
int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y
saxpy_acc(n, a, x, y);
```

Direct Programming: CUDA vs OpenCL

CUDA:

- NVIDIA GPU's Platform
- Platform: Drivers, programming language (CUDA C/C++), API, compiler, debuggers, profilers, ...
- Only NVIDIA GPUs
- Compilation with dedicated compiler (nvcc)
- CUDA fortran

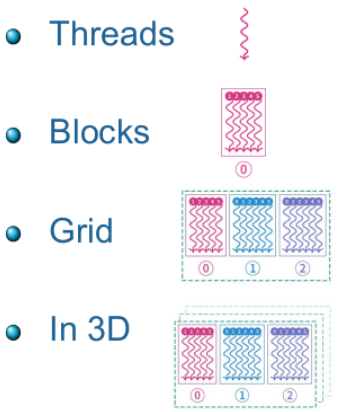
OpenCL:

- Consortium: Open Computing Language by Khronos Group (Apple, IBM, AMD, NVIDIA, ...)
- Programming language (OpenCL C/C++), API, and compiler
- Targets CPUs, GPUs, FPGAs, and other many-core machines
- Fully open source
- Different compilers available

CUDA C/C++

The function running on GPU is called Kernel.

- Access own ID by global variables threadIdx.x, blockIdx.y, ...
- Execution order non-deterministic!
- Only threads in one warp (32 threads of block) can communicate quickly
- A kernel can call other kernels to run on the same GPU (more than one kernel can be executed in the GPU at the same time)
- The kernels exploit the SIMD/SIMT structure of the GPU



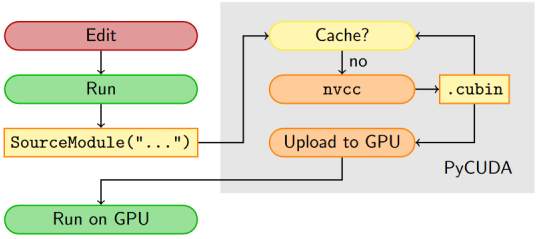
Example

```
__global__ void saxpy_cuda(int n, float a, float * x, float * y) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < n) y[i] = a * x[i] + y[i];
}
int a = 42;
int n = 10;
float x[n], y[n];
// fill x, y
cudaMallocManaged(&x, n * sizeof(float));
cudaMallocManaged(&y, n * sizeof(float));
saxpy_cuda<<<2, 5>>>(n, a, x, y);
cudaDeviceSynchronize();
```

First the data must be copied on the device from the host
Then the kernel is launched
The architecture of threads and blocks is decided at run time

PyCUDA

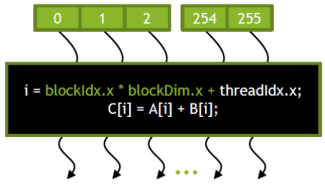
GPUs are everything that scripting languages are not (Highly parallel; very architecture-sensitive; built for maximum throughput).
In this sense GPU and Python can complement each other.
"Alternative" to write the code: Scripting for 'brains' and GPUs for 'inner loops

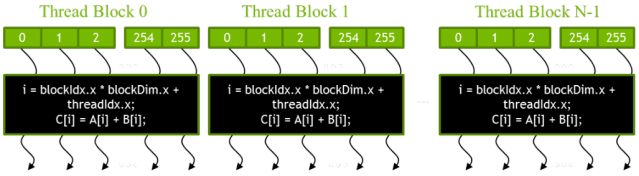


CUDA threads and blocks

A CUDA kernel is executed by a grid of threads.
All threads in a grid run the same code (SIMD or better SPMD (Single Program Multiple Data).
Each thread has indexes that it uses to compute memory addresses and make control decisions.

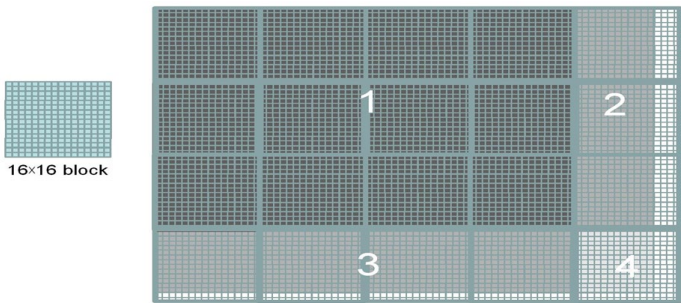
Organize threads in blocks
Threads within a block cooperate via shared memory, atomic operations and barrier synchronization. Threads in different blocks do not interact.





GPU for images

Assume to have a picture of 62x76 pixels.
You want to increase the «luminosity» of each pixel by a factor of 2.

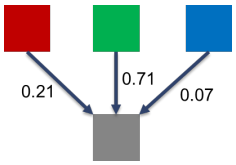



```
__global__ void PictureKernel(float* d_Pin, float* d_Pout,
int height, int width)

{

        // Calculate the row # of the d_Pin and d_Pout element
        int Row = blockIdx.y*blockDim.y + threadIdx.y;

        // Calculate the column # of the d_Pin and d_Pout element
        int Col = blockIdx.x*blockDim.x + threadIdx.x;

        // each thread computes one element of d_Pout if in range
        if ((Row < height) && (Col < width)) {
                d_Pout[Row*width+Col] = 2.0*d_Pin[Row*width+Col];
        }
```

```
// assume that the picture is m × n,
// m pixels in y dimension and n pixels in x dimension
// input d_Pin has been allocated on and copied to device\\
// output d_Pout has been allocated on device
...
dim3 DimGrid((n-1)/16 + 1, (m-1)/16+1, 1);
dim3 DimBlock(16, 16, 1);
PictureKernel<<<DimGrid,DimBlock>>>(d_Pin, d_Pout, m, n);
...z
```

RGB to Grayscale conversion

Assume you want to convert an image in which you have the rgb code for each pixel in greyscale. Rgb is a standard to define the quantity of red, green and blue in each pixel. A greyscale image is an image in which the value of each pixel carries only intensity information.



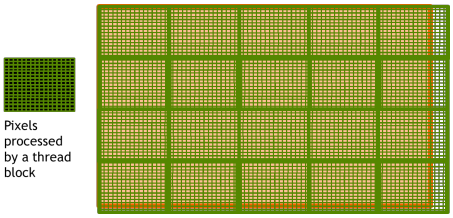
Conversion formula: **For each pixel (I, J) do:** grayPixel[I,J] = 0.21*r + 0.71*g + 0.07*b

```
#define CHANNELS 3 // we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__ void colorConvert(unsigned char * grayImage,
                                                        unsigned char * rgbImage,
                                                int width, int height) {
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        int y = threadIdx.y + blockIdx.y * blockDim.y;

```

```
9   if (x < width && y < height) {
10          // get 1D coordinate for the grayscale image
11          int grayOffset = y*width + x;
12          // one can think of the RGB image having
13          // CHANNEL times columns than the gray scale image
14          int rgbOffset = grayOffset*CHANNELS;
15          unsigned char r = rgbImage[rgbOffset        ]; // red value for pixel
16          unsigned char g = rgbImage[rgbOffset + 2]; // green value for pixel
17          unsigned char b = rgbImage[rgbOffset + 3]; // blue value for pixel
18          // perform the rescaling and store it
19          // We multiply by floating point constants
20          grayImage[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
21   }
22  }
```

Blurring an image

Assume you want to Blur an image.
Defines a Blur box: the blurring is a kind of «average» of the pixel in the blurring box.



la funzione `print` è un po' particolare, è bene usarla solo per il debugging.

il lancio del kernel è sempre asincrono sulla GPU. Se tolgo `cudaDeviceSyncronize()`, la fine del programma (`return 0`) arriva prima dei print.

Vector Sum

Vector Parallel

Abbiamo chiesto di far lavorare solo due thread per blocco! Stiamo sfruttando male la GPU (il coverage delle risorse è molto basso)

Dobbiamo esplicitamente chiedere seè è avvenuto un errore

Chapter 3

Machine Learning

Chapter 4

Fisica Medica

Appendix A

Comandi base di Git e Github

Questa è una breve guida dei principali comandi di Git e GitHub che ho creato a scopo personale. Per farlo ho seguito il video al seguente link.

Creare una Repository partendo da GitHub

Per prima cosa creiamo una nuova repository direttamente dal sito di GitHub. Possiamo creare un file direttamente dall'editor online di GitHub. Sempre dal sito stesso possiamo fare delle modifiche (=commit).

Copiare la Repository in Locale

Apriamo Visual Studio Code e mettiamoci nella cartella dove vogliamo mettere la nostra repository.
A questo punto possiamo aprire il terminal direttamente da VS Code. Dal terminal (usando cd) ci posizioniamo nella cartella dove vogliamo copiare la repository che avevamo creato sul sito di GitHub.

Per farlo, prima di tutto copiamo dal sito di github il link (SSH) alla repository. Sarà una cosa del genere:

```
git@github.com:zaffo1/demo-repo.git
```

A questo punto sul terminale (sempre dentro VS) possiamo scrivere:

```
git clone git@github.com:zaffo1/demo-repo.git
```

Abbiamo appena copiato la repository da GitHub in locale!

Comandi principali in locale

Mettiamoci ora nella repository appena copiata. Possiamo verificare che è una cartella di git perché contiene una sottocartella (nascosta) chiamata ".git". Questa sottocartella è quella che contiene tutti gli update che facciamo ai nostri file.
Per vedere anche le cartelle nascoste (tra cui quella .git) uso il comando:

```
la
```

Non dobbiamo metterci dentro la cartella nascosta chiamata .git, ma rimanere nella cartella che la contiene. E' qui che creeremo i vari file che di cui vorremo tenere traccia.

Per vedere lo stato dei vari file uso il comando:

```
git status
```

Di base git non terrà traccia di tutti i file nella mia cartella, ma devo specificare quali file deve considerare. Per farlo uso il comando **git add**. In particolare, se voglio includere tutti i file uso il punto dopo il comando:

```
git add .
```

Con questo comando ho detto di tener traccia di tutti i file nella cartella.
Altrimenti posso specificare quale file aggiungere a quelli di cui tener traccia, ad esempio:

```
git add nomefile.txt
```

Ricorda: ogni volta che crei un nuovo file devi dire a git di tenerne traccia usando il comando **git add**.

Dopo aver detto a git quali file tracciare, se uso il comand **git status** avrò un output simile al seguente:

```
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md
        new file:   index.html
```

I file ora sono tracciati e sono pronti per essere "committed".
A questo punto posso usare il comando:

```
git commit -m "messaggio"
```

Dove il messaggio dovrebbe contenere informazioni sui cambiamenti che abbiamo effettuato ai nostri file.
Se ho bisogno di aggiungere un ulteriore sottomessaggio posso scrivere invece:

```
git commit -m "messaggio" -m "ulteriore descrizione"
```

A questo punto abbiamo salvato il "commit" in locale, ma ancora non lo abbiamo caricato su GitHub. Per farlo usiamo:

```
git push origin main
```

Creare una Repository in locale

Finora abbiamo lavorato a partire da una repository creata inizialmente su GitHub. Ora vediamo come fare se vogliamo lavorare partendo direttamente in locale.

Creiamo una nuova cartella nella quale vogliamo mettere i nostri file. Per ora è una normalissima cartella, non è una repository di git. Se vogliamo che diventi una cartella di git, da terminale, dopo esserci messi in questa cartella, usiamo il comando:

```
git init
```

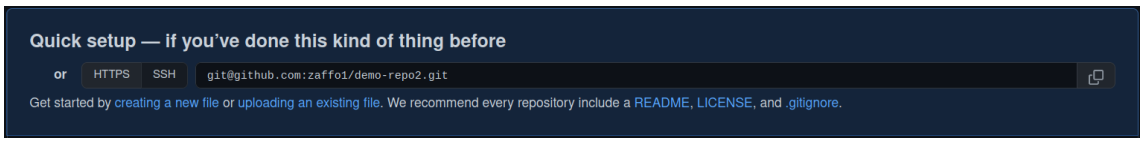
A questo punto possiamo usare tutti i comandi che abbiamo gia visto:

```
git add     #per aggiungere file da tracciare
git status  #per vedere lo stato dei file nella cartella
git commit -m "commento" -m "ulteriore commento"    #per effettuare un "commit"
```

A questo punto vogliamo caricare il tutto su github, come fare? Se usiamo il comando

```
git push origin master
```

otterrò un errore. GitHub non ha idea di dove caricare questa nostra repository locale. La cosa più semplice da fare è quindi creare una repository vuota dal sito di GitHub. Una volta creata uscirà una schermata del genere:



Copiamo il link visualizzato e usiamo il comando:

```
git remote add origin git@github.com:zaffo1/demo-repo2.git
```

per effettuare il collegamento alla repository su GitHub. Possiamo controllare che il collegamento sia avvenuto scrivendo:

```
git remote -v
```

dovremmo ottenere una cosa del genere:

```
origin  git@github.com:zaffo1/demo-repo2.git (fetch)
origin  git@github.com:zaffo1/demo-repo2.git (push)
```

Adesso posso usare il comando:

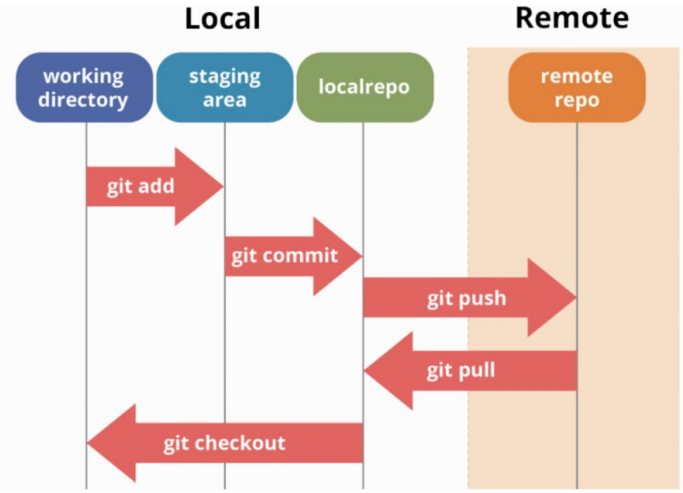
```
git push -u origin master
```

il pezzo "-u" serve per poter impostare di default questa destinazione. Cosicché le prossime volte mi basterà scrivere

```
git push
```

Abbiamo caricato la nostra repository su GitHub!
Nota: a volte ho usato **main** e a volte **master**. Bisogna essere consistenti.

Git Workflow



Appendix B

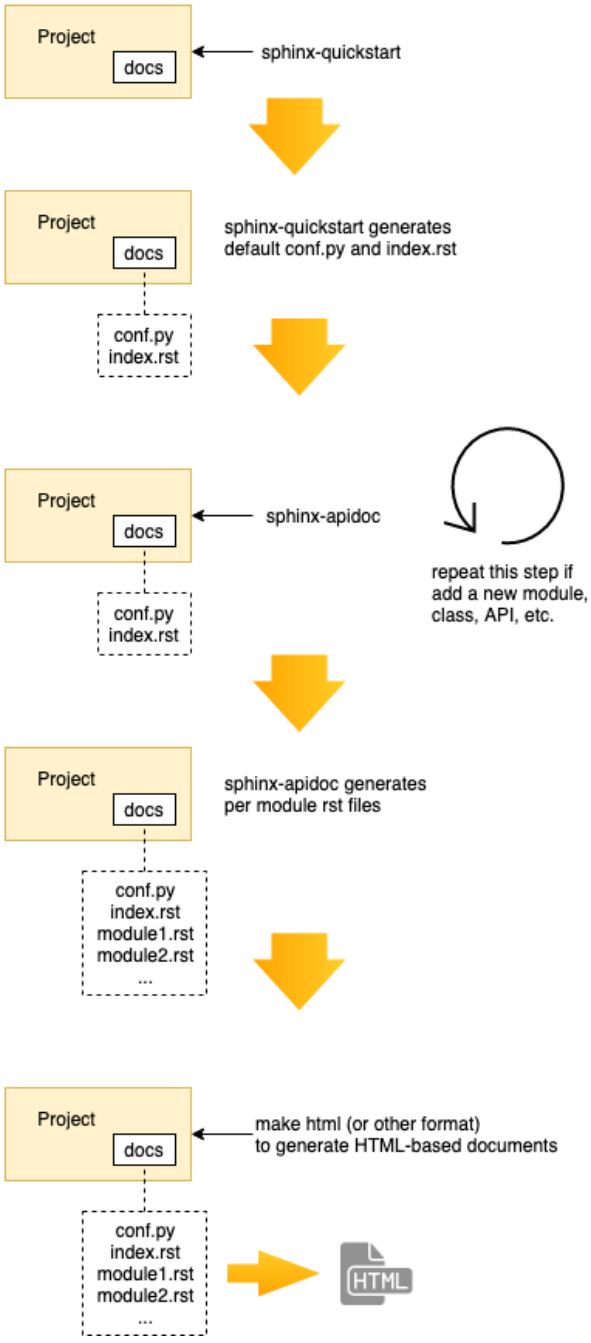
Usare Sphinx per creare la documentazione

Nota: ho seguito fondamentalmente le istruzioni scritte nei tutorial ai seguenti link:
`https://shunsvineyard.info/2019/09/19/use-sphinx-for-python-documentation/`
`https://towardsdatascience.com/documenting-python-code-with-sphinx-554e1d6c4f6d`

Sphinx provides two command-line tools: sphinx-quickstart and sphinx-apidoc.

1. `sphinx-quickstart` sets up a source directory and creates a default configuration, conf.py, and a master document, index.rst, which serves as a welcome page of a document.
2. `sphinx-apidoc` generates reStructuredText files to document from all found modules.

Il workflow tipico può essere rapprensetato nel seguente diagramma:



Step 1: Use sphinx-quickstart to generate Sphinx source directory with conf.py and index.rst

Supponiamo che il nostro progetto abbia la seguente struttura:

```
sphinx_basics
 |- docs
 |- maths
    |- add.py
    |- divide.py
    |- multiply.py
    |- subtract.py
    |- __init__.py
```

Vogliamo mettere tutta la documentazione nella cartella `docs`.
Per farlo ci mettiamo nella cartella `docs` e scriviamo:

```
sphinx-quickstart
```

Ci varranno chieste alcune domande riguardo al nostro progetto. In particolare, rispondere `(y)` alla domanda:

```
> Separate source and build directories (y/n) [n]: y
```

A questo punto la cartella `docs` avrà una struttura del genere:

```
docs
-- Makefile
-- build
-- make.bat
-- source
    |- _static
    |- _templates
    |- conf.py
    |- index.rst
```

Step 2: Configure the conf.py

sphinx-quickstart generates a few files, and the most important one is conf.py, which is the configuration of the documents. Although conf.py serves as a configuration file, it is a real Python file. The content of conf.py is Python syntax.

Go to your conf.py file and uncomment line numbers 13,14 and 15. Change the `os.path.abspath('.')` to `os.path.abspath('..')`. Here, we tell sphinx that the code is residing outside of the current docs folder.

Per far funzionare le cose, puoi direttamente copiare e incollare il seguente script:

```
# Configuration file for the Sphinx documentation builder.
#
# This file only contains a selection of the most common options. For a full
# list see the documentation:
# https://www.sphinx-doc.org/en/master/usage/configuration.html

# -- Path setup --------------------------------------------------------------

# If extensions (or modules to document with autodoc) are in another directory,
# add these directories to sys.path here. If the directory is relative to the
# documentation root, use os.path.abspath to make it absolute, like shown here.
#
import os
import sys
sys.path.insert(0, os.path.abspath('..'))


# -- Project information -----------------------------------------------------

project = 'PDF random generator'
copyright = '2022, Lorenzo Zaffina'
author = 'Lorenzo Zaffina'

# The full version, including alpha/beta/rc tags
release = '00.00.01'

```

```
27
28  # -- General configuration ------------------------------------------------
29
30  # Add any Sphinx extension module names here, as strings. They can be
31  # extensions coming with Sphinx (named 'sphinx.ext.*') or your custom
32  # ones.
33  extensions = [
34      'sphinx.ext.autodoc',
35      'sphinx.ext.viewcode',
36      'sphinx.ext.napoleon'
37  ]
38
39  # Add any paths that contain templates here, relative to this directory.
40  templates_path = ['_templates']
41
42  # List of patterns, relative to source directory, that match files and
43  # directories to ignore when looking for source files.
44  # This pattern also affects html_static_path and html_extra_path.
45  exclude_patterns = ['_build', 'Thumbs.db', '.DS_Store']
46
47
48  # -- Options for HTML output ----------------------------------------------
49
50  # The theme to use for HTML and HTML Help pages.  See the documentation for
51  # a list of builtin themes.
52  #
53  html_theme = 'sphinx_rtd_theme'
54
55  # Add any paths that contain custom static files (such as style sheets) here,
56  # relative to this directory. They are copied after the builtin static files,
57  # so a file named "default.css" will overwrite the builtin "default.css".
58  html_static_path = ['_static']
```

Step 3: Use sphinx-apidoc to generate reStructuredText files from source code

A questo punto, andiamo nella cartella madre del nostro progetto (`sphinx_basics`) e scriviamo (con le sostituzioni opportune):

```
1  sphinx-apidoc -f -o <path-to-output> <path-to-module>
```

-f means force overwriting of any existing generated files.
-o means the path to place the output files.

Nel caso preso in esempio, scriviamo:

```
1  sphinx-apidoc -f -o docs maths/
```

Step 4: Including module.rst and generating html

The generated `modules.rst` contains all the modules. So we need to add the `modules.rst` to `index.rst`.

Apriamo il file (`index.rst`) e scriviamo `modules` come nel seguente esempio:

```
1  .. toctree::
2     :maxdepth: 2
3     :caption: Contents:
```

```
4
5       modules
```

A questo punto tutto è pronto per generare la nostra documentazione. Andiamo nella cartella `docs` e scriviamo:

```
1   make html
```

Ecco fatto! Abbiamo generato la nostra documentazione nella cartella `_build`