Laboratorio di web scraping
AA. 2024-2025

docente: Laura Ricci

laura.ricci@unipi.it

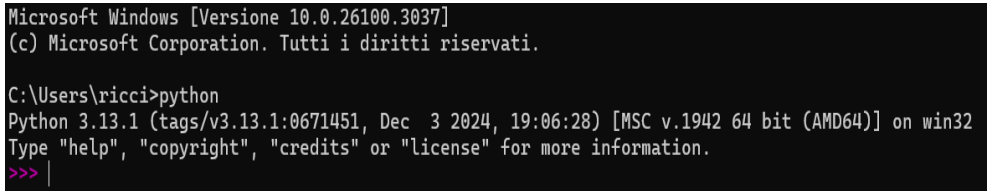
Lezione 2
Python – concetti generali
31/1/2025

PYTHON: VERSATILITA'

- utilizzato in molti contesti
  - data analysis e visualization
  - machine learning
  - scripting
  - web development
  - software generale
- versatilità: utilizzabile da utenti con background diversi
  - economisti, statistici, scienziati senza specifica formazione informatica
- diversi contesti richiedono diversi livelli di competenza/conoscenza del linguaggio
- in questo corso
  - analizzeremo le funzionalità principale del linguaggio
  - in generale, ne utilizzaremo un sottoinsieme

CARATTERISTICHE PRINCIPALI

- tipizzazione dinamica forte
- object oriented
- interpretato
- interattivo
  - interprete CLI dei comandi



- modulare
- ampia disponibilità di librerie
- facilmente interfacciabile con altri linguaggi (librerie efficienti C++)
- molti strumenti utli
  - pip: gestore di pacchetti Python, facile istallazione e deistallazione di moduli e librerie

ISTALLAZIONE DI PACCHETTI

- Python ha il suo gestore di pacchetti `pip`

```
Usage:
    pip <command> [options]
commands:
    install Install packages.
    download Download packages.
    uninstall Uninstall packages.
    freeze Output installed packages in requirements format.
    list List installed packages.
    show Show information about installed packages.
    search Search PyPI for packages.
    wheel Build wheels from your requirements.
    hash Compute hashes of package archives.
    help Show help for commands.
```

CARATTERISTICHE PRINCIPALI DEL LINGUAGGIO

- indentazione per delimitare blocchi di codice, invece di `{ }` o `begin...end`
- strong e dynamic typing
- exception handling simile a `JAVA`
- Diverse collezioni
  - liste
  - tuples
  - dictionaries: simili a hash map di `JAVA`
  - sets
- higher-order functions
- iterators
- name resolution: `LEGB` scoping (local, encolsed, global, built-in)

VARIABILI

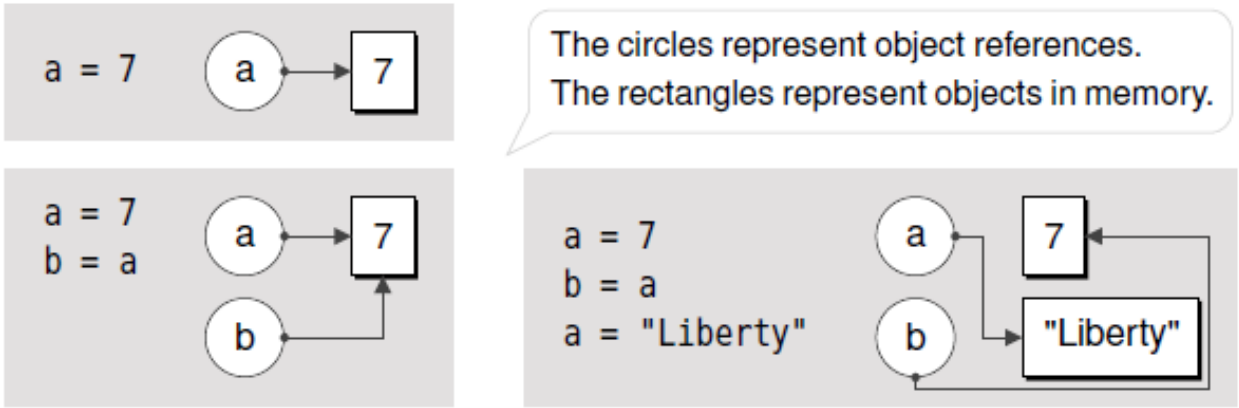
- una variabile inizia ad esistere quando si assegna un valore ad un nome, non esiste alcuna dichiarazione esplicita di variabile

```
>>> a = 10
```

- dinamicamente si può cambiare valore e tipo:

```
>>> a = 10.1
>>> a = "Stringa"
```

- ogni variabile contiene un riferimento ad un oggetto, qualsiasi cosa in Python è un oggetto

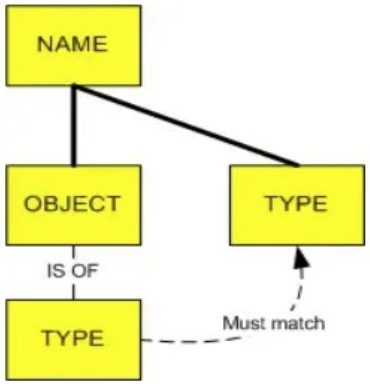


STATIC TYPING

- dichiarazioni di variabili che definiscono e "fissano" il tipo della variabile a compile time
  - Java, C++,...
- una volta fissato il tipo non è più possibile cambiarlo
  - a quella variabile verranno sempre assegnati valori di quel tipo
  - altrimenti segnalato un errore a compile time

```
int data;
data = 0;
data = "Static Typing"; // causes an compilation error
```

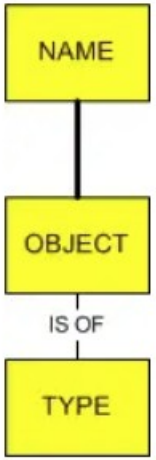
- vantaggi
  - errori segnalati a compile time
  - codice ottimizzato
  - maggiori performances



PYTHON: DYNAMIC TYPING

- il programmatore non deve dichiarare il tipo di una variabile prima di usarla
- variabile
  - nome legato dinamicamente ad un oggetto(valore)
  - l'oggetto ha un tipo, ma la variabile non è legata al tipo di quell'oggetto, può essere riassegnata a oggetti di tipo di verso
- l'interprete del linguaggio determina il tipo della variabile e run time a seconda del contesto

```
data=1
data='Hello World' //no error
data= false // no error
```



- duck typing

  "*when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck*"

WEAK/STRONG TYPING

- la distinzione riguarda il comportamento del linguaggio riguardo alle istruzioni che coinvolgono valori di tipo diverso
- `strongly typed languages`, in presenza di dati di tipo diverso
  - in genere generano un errore (a compile time o a run time)
  - il programmatore deve inserire esplicitamente una operazione di conversione del tipo per evitare l'errore
- `weakly typed languages`, in presenza di dati di tipo diverso
  - il supporto può eseguire conversioni implicite di tipo
  - talvolta il risultato è inatteso
- `Java Script` è weakly typed

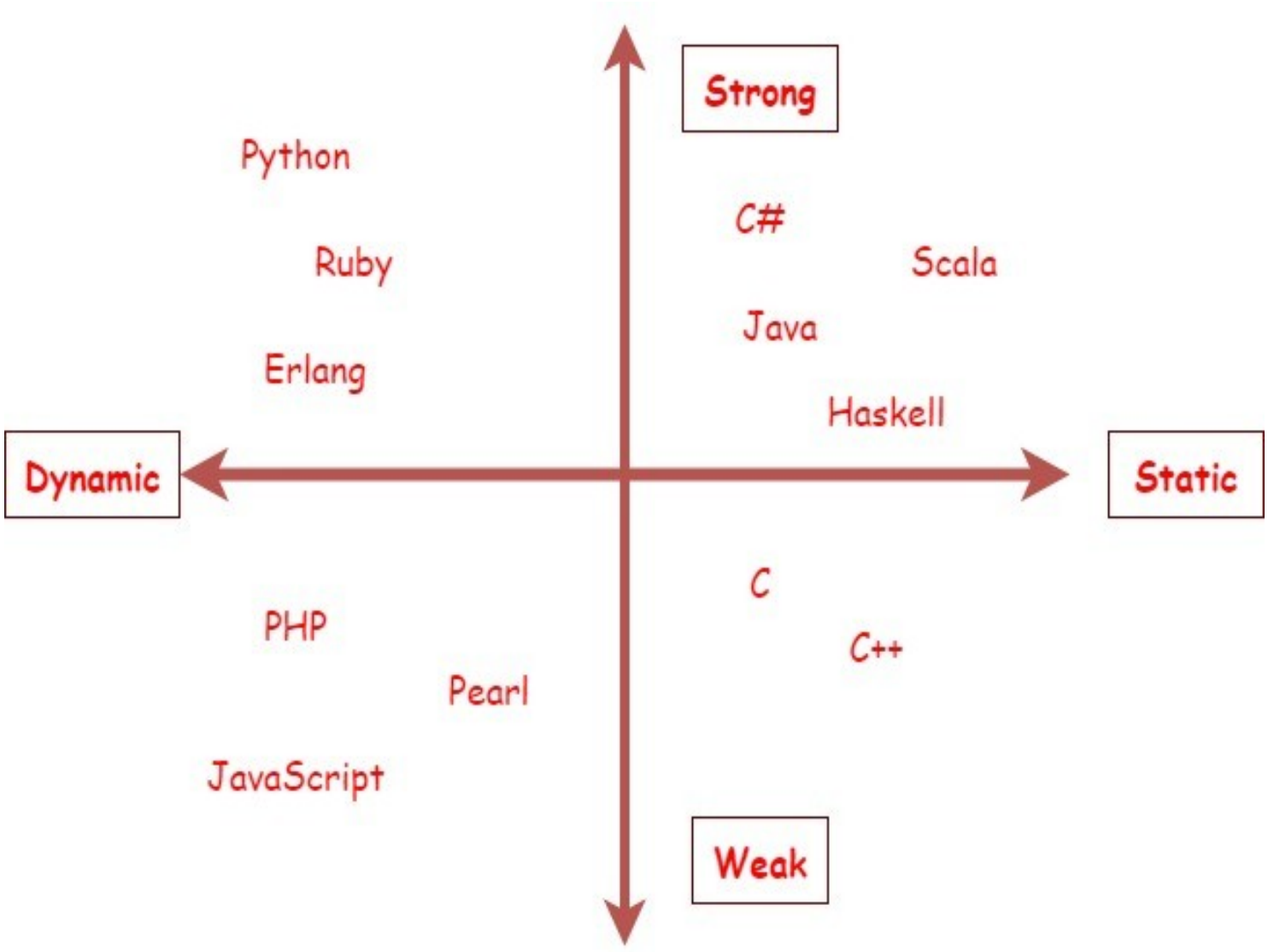
```
> a = 10
> b = "K"
> a + b
'10K'
```

PYTHON: DYNAMIC E STRONG TYPED

```
In [2]: number = 3
        print ( number , type ( number ))
        print ( number + 42)
        number = "3"
        print ( number , type ( number ))
        print ( number + 42)
```

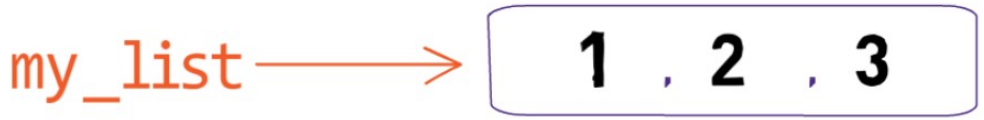
```
3 <class 'int'>
45
3 <class 'str'>

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[2], line 6
      4 number = "3"
      5 print ( number , type ( number ))
----> 6 print ( number + 42)

TypeError: can only concatenate str (not "int") to str
```

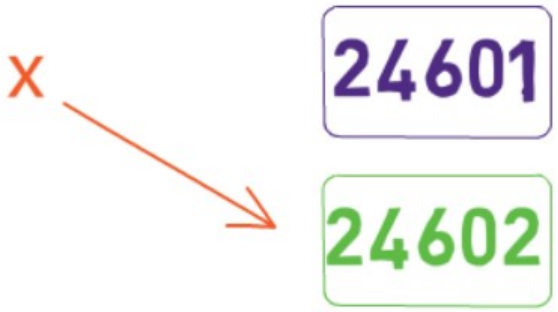
STRONG/WEAK DYNAMIC/STATIC LANGUAGES

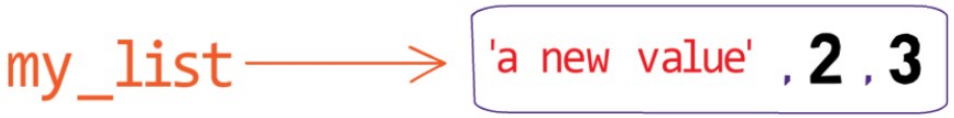


OGGETTI MUTABILI ED IMMUTABILI

- si parla di oggetto `mutabile` quando il suo valore può essere alterato
- un oggetto che non può essere alterato dopo la creazione è detto `immutabile`
- mutabilità dei tipi di dati in `Python`:
  - i tipi semplici sono tutti immutabili
  - i tipi strutturati possono essere mutabili o non mutabili

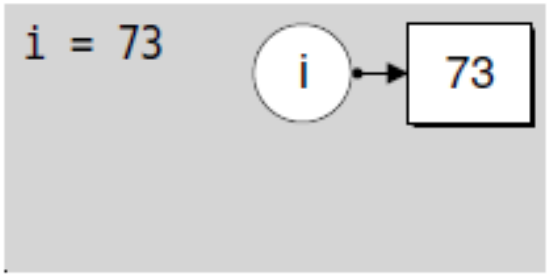


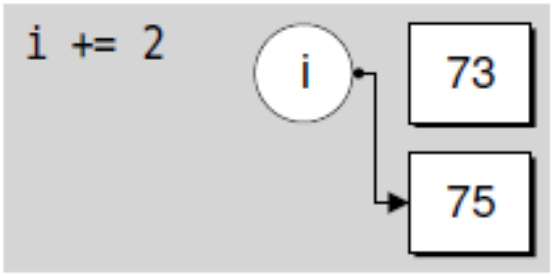




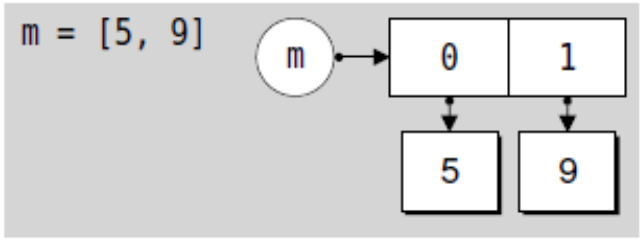
OGGETTI IMMUTABILI E MUTABILI

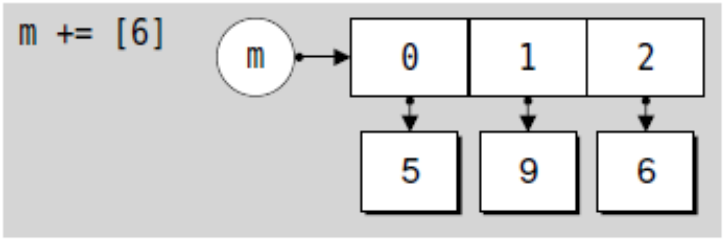
- il tipo di dati `int` è immutabile!
- anche l'operazione di incremento crea un nuovo oggetto e lega la variabile ad esso





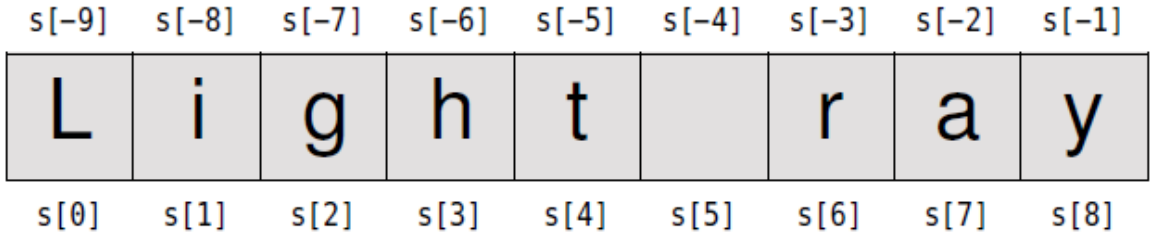
- le liste sono mutabili
- non è necessario legare la variabile ad un nuovo oggetto





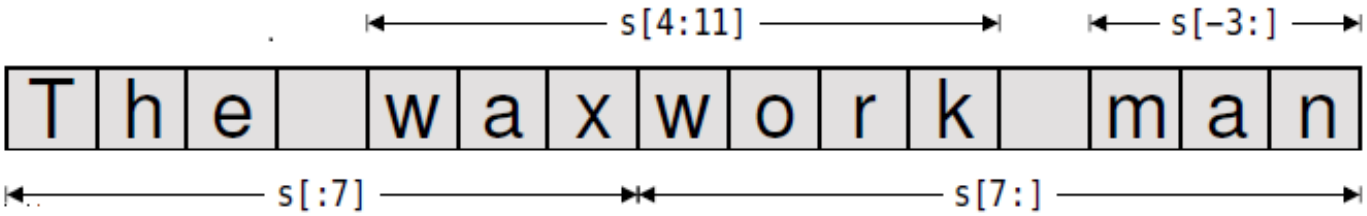
TIPI PRIMITIVI STRINGHE

- le stringhe sono delle sequenze (come liste e tuple) e sono immutabili
- ogni elemento in una sequenza può essere acceduto mediante l'operatore [ ]

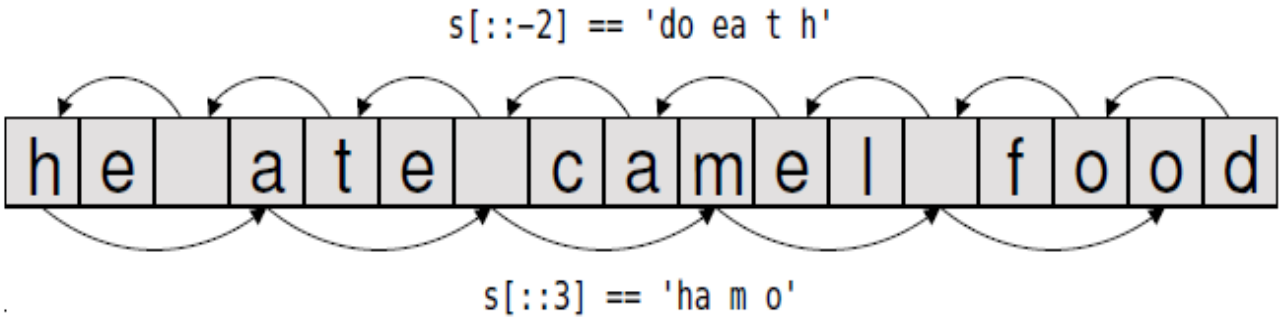


- notare l'uso di indici negativi, utile in special modo il -1 che da accesso all'ultimo carattere della stringa
- su tutti gli oggetti di tipo sequenza è possibile definire due operazioni molto utili
  - slicing
  - striding

SLICING E STRIDING



slicing



striding

I TIPI DI DATO STRUTTURATI

LISTS VS. TUPLES VS. SETS VS. DICTIONARIES

	Lists	Tuples	Sets	Dictionaries
Ordering	Ordered	Ordered	Unordered	Ordered Unordered before Python 3.7
Indexing	Indexed	Indexed	Not Indexed	Keyed
Mutability	Mutable	Immutable	Mutable Only Adding and Removing	Mutable
Duplicates Allowed	Yes	Yes	No	Yes Only in values and not in keys
Types Allowed	Mutable and Immutable	Mutable and Immutable	Only Immutable	Only Immutable In keys

CONTROLLO: L'IDENTAZIONE

- indentazione per delimitare blocchi di codice, invece di `{ }` o `begin...end`
- tutte le istruzioni che presentano lo stesso numero di spazi di indentazione appartengono allo stesso blocco: non solo una scelta stilistica, ma semantica del linguaggio

```
i = 0
while(10 > i):
    i += 1
    print(i)
```

- attenzione agli errori!

```
def greet(name):
print("Hello,", name)  # This will raise an IndentationError
```

- errori tipici: mischiare spazi bianchi con tab: usare sempre o gli uni o gli altri

CONTROLLO: L'ISTRUZIONE IF

- è il controllo condizionale unificato di Python
- oltre ad implementare il classico costrutto *if-then-else* è anche un equivalente del costrutto *switch* presente in altri linguaggi
- sintassi:

```
if expression:
   statement(s)
elif expression:
   statement(s)
elif expression:
   statement(s)
...
else:
   statement(s)
```

CONTROLLO: L'ISTRUZIONE WHILE

- l'istruzione `while` definisce un ciclo iterativo basato su una condizione
- il ciclo continua fino a quando la condizione è `True`
- sintassi:

```
while expression:
        statement(s)
[else:
        statement(s)]
```

- il blocco `else` è opzionale e viene eseguito quando la condizione del ciclo diventa `False`
- se invece il ciclo viene interrotto esplicitamente (istruzione `break`), e non perchè la condizione è falsa, allora il blocco `else` non viene eseguito

CONTROLLO: L'ISTRUZIONE FOR

- l'istruzione `for` definisce un ciclo iterativo ripetuto per tutti gli elementi di un'espressione `iterabile`
- sintassi:

```
for target in iterable:
    statement(s)
[else:
    statement(s)]
```

- il blocco `else` è opzionale e viene eseguito dopo la conclusione dell'iterazione
  - se il ciclo viene interrotto esplicitamente da un'istruzione `break` allora il blocco `else` non viene eseguito
- durante il for loop non è possibile alterare l'oggetto `iterabile` su cui si effettuano le iterazioni

CONTROLLO: L'ISTRUZIONE FOR

- si può iterare con degli indici:

```
elenco = ['uno', 'due', 'tre']
for i in range(0, len(elenco)):
      print(elenco[i])
```

- ma è più naturale iterare direttamente sulla lista:

```
elenco = ['uno', 'due', 'tre']
for elemento in elenco:
    print(elemento)
```

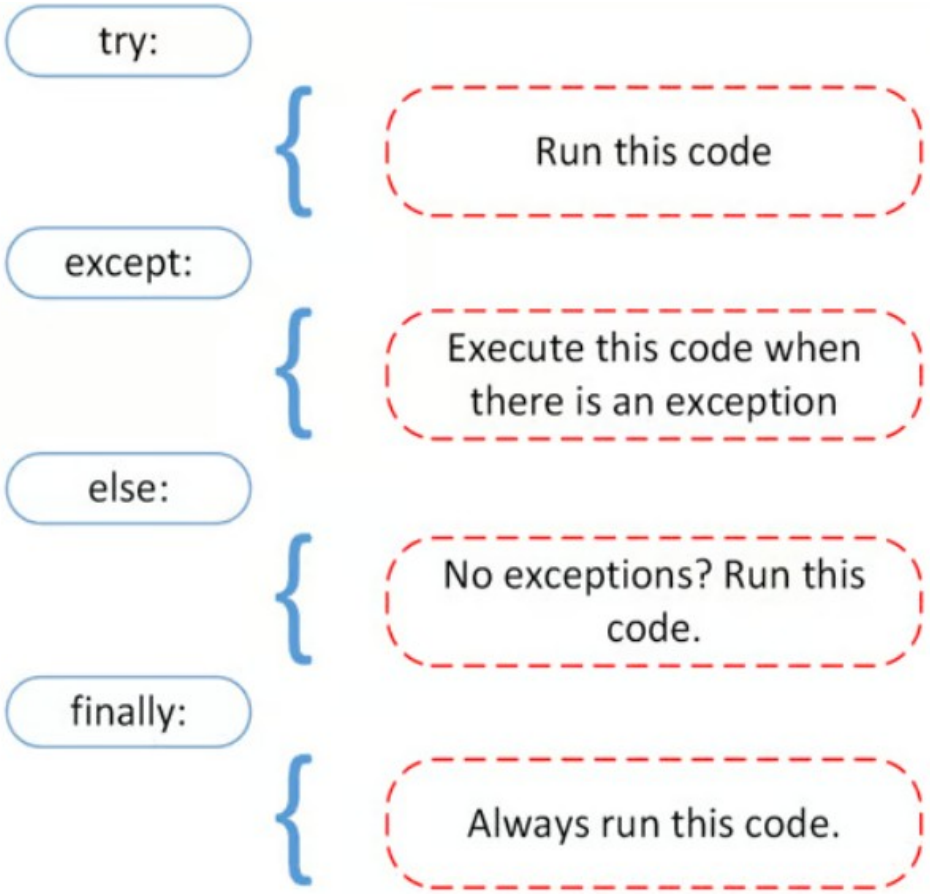
- il range ha senso quando è necessario produrre la sequenza numerica:

```
for valore in range(1, 10):
    print(valore)
```

- sui dizionari l'iterazione avviene sulla chiave:

```
tabella = {'primo': 20,'secondo': 10,'terzo': 5}
for chiave in tabella:
      print(tabella[chiave])
```

EXCEPTION HANDLING: FORMA GENERALE



EXCEPTION HANDLING: TRY/EXCEPT

- molto simile a JAVA:

```
s = input("enter an integer: ")
try:
   i = int(s)
   print("valid integer entered:", i)
except ValueError as err:
   print(err)
```

- la parte as...è facoltativa, utile solo se si vogliono avere informazioni specifiche sull'evento che ha generato l'eccezione
- possibile intercettare più eccezioni nello stesso blocco except
- si può inserire un blocco finally, sempre eseguito

EXCEPTION HANDLING: TRY/EXCEPT

- molto simile a JAVA:

```
s = input("enter an integer: ")
try:
   i = int(s)
   print("valid integer entered:", i)
except ValueError as err:
   print(err)
```

- se l'eccezione non viene intercettata dagli except del blocco allora viene propagata
- la propagazione avviene lungo lo stack di chiamata, fino a quando non c'è un blocco except attivo adatto
- se non c'è il blocco l'eccezione arriva all'interprete che ritorna l'eccezione a livello di console

EXCEPTION HANDLING: TRY/EXCEPT

```
import sys
try:
   f = open('myfile.txt')
   s = f.readline()
   i = int(s.strip())
except IOError as e:
   print ("I/O error({0}): {1}".format(e.errno,e.strerror))
except ValueError:
   print ("Could not convert data to an integer.")
except:
   print ("Unexpected error:", sys.exc_info()[0])
   raise
```

- sono ammessi più blocchi except
- l'ultimo except può omettere la specifica dell'eccezione: ha il significato di una *wildcard*
- rilancio dell'eccezione con istruzione raise

EXCEPTION HANDLING

- Python è un linguaggio ad oggetti!
- definita una gerarchia di 'oggetti eccezione'
- `BaseException` è la classe base di tutte le eccezioni
- attenzione a intercettare per prime le eccezioni "più specifiche"

```
BaseException
 ├── BaseExceptionGroup
 ├── GeneratorExit
 ├── KeyboardInterrupt
 ├── SystemExit
 └── Exception
      ├── ArithmeticError
      │    ├── FloatingPointError
      │    ├── OverflowError
      │    └── ZeroDivisionError
      ├── AssertionError
      ├── AttributeError
      ├── BufferError
      ├── EOFError
      ├── ExceptionGroup [BaseExceptionGroup]
      ├── ImportError
      │    └── ModuleNotFoundError
      ├── LookupError
      │    ├── IndexError
      │    └── KeyError
      ├── MemoryError
      ├── NameError
      │    └── UnboundLocalError
      ├── OSError
      │    ├── BlockingIOError
      │    ├── ChildProcessError
      │    ├── ConnectionError
      │    │    ├── BrokenPipeError
      │    │    ├── ConnectionAbortedError
      │    │    ├── ConnectionRefusedError
      │    │    └── ConnectionResetError
      │    ├── FileExistsError
      │    ├── FileNotFoundError
      │    ├── InterruptedError
      │    ├── IsADirectoryError
      │    ├── NotADirectoryError
      │    ├── PermissionError
      │    ├── ProcessLookupError
      │    └── TimeoutError
      ├── ReferenceError
      ├── RuntimeError
      │    ├── NotImplementedError
      │    ├── PythonFinalizationError
      │    └── RecursionError
      ├── StopAsyncIteration
      ├── StopIteration
      ├── SyntaxError
      │    └── IndentationError
      │         └── TabError
      ├── SystemError
      ├── TypeError
      ├── ValueError
      │    └── UnicodeError
      │         ├── UnicodeDecodeError
      │         ├── UnicodeEncodeError
      │         └── UnicodeTranslateError
      └── Warning
           ├── BytesWarning
           ├── DeprecationWarning
           ├── EncodingWarning
           ├── FutureWarning
           ├── ImportWarning
           ├── PendingDeprecationWarning
           ├── ResourceWarning
           ├── RuntimeWarning
           ├── SyntaxWarning
           ├── UnicodeWarning
           └── UserWarning
```

LA CLAUSOLA ELSE

```
x = 1
try:
    print(5 / x)
except ZeroDivisionError:
    print("I am the except clause!")
else:
    print("I am the else clause!")
finally:
    print("I am the finally clause!")

print("I am executing after the try clause!")

# 5.0
# I am the else clause!
# I am the finally clause!
# I am executing after the try clause!
```

```
x = 0
try:
    print(5 / x)
except ZeroDivisionError:
    print("I am the except clause!")
else:
    print("I am the else clause!")
finally:
    print("I am the finally clause!")

print("I am executing after the try clause!")

# I am the except clause!
# I am the finally clause!
# I am executing after the try clause!
```

Proviamo l'esecuzione con l'interprete interattivo!

INTERAGIRE CON L'INTERPRETE

- invocare il comando `python` da shell

```
C:\Users\ricci>python
Python 3.8.10 (tags/v3.8.10:3d8993a, May  3 2021, 11:48:03) [MSC v.1928      64
bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- `>>>` prompt dell'interprete
- immetto comandi che vengono eseguiti immediatamente dall'interprete

```
>>> cost = 27.00
>>> taxrate = .075
>>> cost * taxrate
2.025
```

- nel caso si immetta un costrutto composto, l'interprete eseguirà il comando solo quando è completato il codice del costrutto
  - `prompt cambia da >>> a …`
  - … il comando deve essere completato

ESEGUIRE UN MODULO COME SCRIPT

- soluzione non interattiva
- inserire tutto il codice in un modulo memorizzato in un file (estensione `.py`) con un editor più o meno sofisticato, e poi eseguire il codice invocando `python` da riga di comando

```
s = 'Hello world'
print(s)
```

my_program.py

- invocare l'interprete da riga di comando per l'esecuzione del modulo

```
> python my_program.py
Hello Wordle
```

- l'interprete esegue le istruzioni sequenzialmente
- non c'è una fase di compilazione (niente di analogo a `javac`,...)
- adatto per esecuzione di scripts

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

- `somma` è l'identificatore della funzione
- `a` e `b` sono i parametri formali della funzione e non hanno una specifica di tipo (*dynamic typing*)
- valore/i di ritorno:
  - l'istruzione return permette di ritornare uno o più valori come risultato della funzione
  - i valori possono essere di un tipo qualsiasi, mutabile o immutabile, raggruppati eventualmente in un contenitore (es. lista) o in una tupla qualora li si separi semplicemente con delle virgol
  - senza un return esplicito di un valore la funzione ritorna `None`

TIPIZZAZIONE DINAMICA

- la funzione somma definita in precedenza può essere chiamata con argomenti di tipo diverso:

```
>>> somma(2, 5)
7
>>> somma("super", "man")
'superman'
>>> somma(["a", 1, False], [5, 29, None])
['a', 1, False, 5, 29, None]
```

- errori vengono generati se non è definito l'operatore di somma o se gli operandi non sono compatibili

```
>>> somma({"a", 1, False}, {5, 29, None})
Traceback (most recent call last):
…..
 …..
TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

FUNZIONI COME OGGETTI

- le funzioni possono essere assegnate come fossero oggetti mutabili:

```
>>> somma(1, 4)
5
>>> f = somma
>>> f(3, 9)
12
```

- le funzioni possono essere passate come parametri ad altre funzioni:

```
>>> def cubo(x):
... return x**3
...
>>> def chiama(f, arg):
... return f(arg)
...
>>> chiama(cubo, 3)
27
```

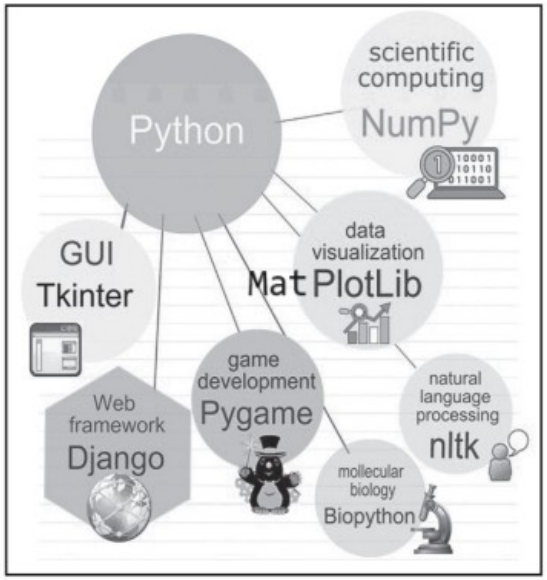
PYTHON: MODULI

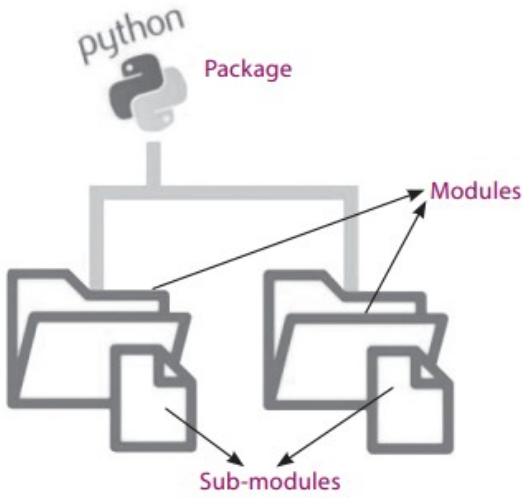
- un programma Python è tipicamente costituito da più file sorgenti, detti anche `moduli`
- ogni modulo può avere la necessità di utilizzare funzioni, classi o variabili globali definite in altri moduli
- per gestire queste dipendenze in altri moduli si usano i costrutti *import*

```
import modname as varname
```

`varname` è il nome con cui il modulo importato è accessibile all'interno del modulo che lo importa

- I moduli possono essere organizzati e raggruppati gerarchicamente in packages
- sub-module che sono le le funzioni appartenenti ad un modulo





PYTHON: IMPORTAZIONE DI MODULI

- per creare un modulo Python, è sufficiente salvare il codice Python in un file con estensione `.py`

main.py

```
1
2  # my_module.py
3
4  def greet(name):
5      print("Hello, " + name)
6
7  def square(y):
8      return y ** 2
9
10 pi = 3.1416
```

- per importare un modulo

main.py

```
1  import my_module
2
3  my_module.greet("Alice")
4  print(my_module.square(5))
5  print(my_module.pi)
```

- il nome `x` del modulo `m` è un attributo di `m` è accessibile con la notazione `m.x`

PYTHON: NAMESPACES

- un namespace è un mapping da nomi a oggetti
  - implementato come un dizionario
  - dato un nome (variabile, metodo), `Python` utilizza il dizionario per reperire l'oggetto associato a quel nome
- per conoscere tutti i nomi in un modulo, è possibile usare il metodo `dir()`

```
>>> import math
>>> math
<module 'math' (built-in)>
>>> dir(math)
['__doc__', '__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']
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.pow(3, 5)
243.0
```

PYTHON: LA VARIABILE __NAME__

- in ogni modulo sono definite alcune variabili speciali
- variabile `__name__` : il suo valore dipende dal contesto di esecuzione del modulo
  - se il codice del modulo viene invocato come entry point del programma, la variabile contiene il valore `__main__`.
  - se il modulo invece viene importato da un altro modulo, la variabile `__name__` contiene il nome del modulo, che è il nome del file che lo contiene
  - quando può essere utile?
    - per creare degli *unit test* per la verifica del corretto funzionamento del modulo
    - per capire se passare al modulo degli argomenti da riga di comando

PYTHON: LA VARIABILE __NAME__

- nel caso si scriva un modulo che poi può essere utilizzato da altri, è bene inserire degli unit test per testarne il comportamento
- tali test non devono però essere eseguiti se il modulo e importato da altri

```
if __name__ == '__main__':
    assert ...
    assert ...
    assert ...
    print('All tests passed.')
```

PYTHON: LA VARIABILE __NAME__

- supponiamo di definire un modulo che include la definizione di due funzioni per il calcolo dei numeri di `Fibonacci`

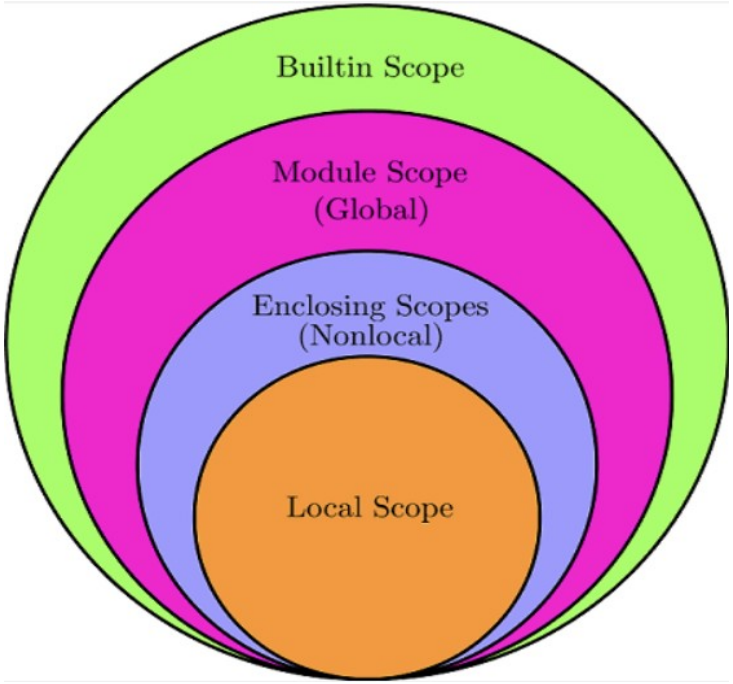
```python
# File fibo.py - Fibonacci numbers module

import sys

def fib(n): # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
        print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    print(result)

if __name__ == "__main__":
    num=int(sys.argv[1])
    fib2(num)
```

```
>>> python fiboMain.py 60
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

- eseguito con `__name__ impostato a __main__`
- consente di reperire argomenti passati da linea di comando (come in `JAVA`)

PYTHON: LEGB NAMESPACES

- *scope:* insieme di nomi associati ad un particolare ambiente
- un nome può essere presente in più scope: importante conoscere a quale nome si fa riferimento
- gli scope in `Python`



PYTHON: LEGB NAMESPACES

- scope: regione del programma in cui il riferimento alla variabile è accessibile
  - l'interprete va a cercare il valore della variabile nello scope in cui è definita
- durante l'esecuzione del programma esistono 4 namespace, ed in essi viene cercato il binding di una variabile nell'ordine seguente
  - scope locale: variabili locali a una funzione
  - enclosing scope: scope delle funzioni che racchiudono la funzione dove si trova la variabile, contiene variabili non locali
  - global scope: lo scope contenente le variabili del modulo
  - built-in scope: contiene nomi built-in
- accesso per default avviene nello scope locale, però le variabili degli altri scope possono essere riferite con le `keywords nonlocal` o `global`

PYTHON: LEGB NAMESPACES

In [9]:

```
def flower_test():
    def do_local():
        flower = "local_rose"
    def do_nonlocal():
        nonlocal flower
        flower = "nonlocal lily"
    def do_global():
        global flower
        flower = "global cyclamen"
    flower = "daffodils"
    do_local()
    print("After local assignment:", flower) # not affected
    do_nonlocal()
    print("After nonlocal assignment:", flower) # affected
    do_global()
    print("After global assignment:", flower) # not affected
flower_test()
print("In global scope:", flower)

```

```
After local assignment: daffodils
After nonlocal assignment: nonlocal lily
After global assignment: nonlocal lily
In global scope: global cyclamen
```

PYTHON: LEGB NAMESPACES

```
[10]: def test(x):
          print(x)
          for x in range(5):
              print(x)
          print(x)
```

```
[11]: test("Hello!")
```

```
Hello!
0
1
2
3
4
4
```

- notare che, a differenza di altri linguaggi, le strutture di controllo non introducono un nuovo scope

PYTHON: CONCORRENZA

- l'interprete assicura che solo un thread alla volta può eseguire il bytecode di Python, grazie alla "Global Interpreter Lock"
- un thread deve possedere questa lock prima di accedere a un qualsiasi oggetto Python
  - lock rilasciata quando il thread esegue I/O
  - implementazione dell'interprete semplificata: un accesso alla volta
  - bootleneck, degradazione di performance
- soluzioni non soddisfacenti per task CPU-bound
- alcune estensioni per rilascio di lock quando il thread esegue task computationally intensive, come hashing e compression
- vedremo un esempio nella lezione successiva

COME ESEGUIRE PYTHON

- diverse modalità per interagire con l'interprete `Python`
  - eseguire moduli (script) non interattivamente
  - interagire con l'interprete
    - direttamente da console
    - mediante una "console avanzata": `IPython, Interactive Python Shell`
    - utilizzando un notebook

INTERAGIRE CON L'INTERPRETE

```
>>> for x in range(10):
...     print(x)
...
0
1
2
3
4
5
6
7
8
9
```

- quando ho finito il corpo del loop inserisco un return (secondo prompt …) indicando all'interprete che il corpo del ciclo è terminato
- l'interprete esegue il comando

INTERAGIRE CON L'INTERPRETE

- aprire una console `IPython` per intergire con l'interprete
- fornisce funzionalità aggiuntive
  - esplorare le variabili
  - richiamare moduli ed eseguirli
  - e tanti altri comandi.....

```
In [1]: %run my_file.py
Hello world

In [2]: s
Out[2]: 'Hello world'

In [3]: %whos
Variable   Type    Data/Info
--------------------------
s          str     Hello world
```

AMBIENTE AVANZATO: JUPYTER NOTEBOOK

- notebook interattivi
  - equivalente "moderno" del tradizionale notebook
- `Jupyter`
  - web application
  - portmanteau di `Julia`, `Python` e `R`, i tre linguaggi per cui è stato progettato
  - spin-off di `IPython, Interactive Python Shell` rebranded per essere utilizzato con linguaggi diversi
- un notebook interattivo open source basato su celle che possono contenere:
  - codice
  - testo: `Markdown`
    - un linguaggio di markup (come `HTML`, `XML`, `Latex`) ma lightweight
  - visualizzazione di tabelle e grafi
  - supporta anche `HTML` e Latex
- `https://jupyter.org/try`

JUPYTER NOTEBOOK: INTERFACCIA WEB

IP[y]: Notebook Modulation Last Checkpoint: Jan 05 11:01 (autosaved)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

An angle modulated signal generally can be written as

$u(t) = A_c cos(2\pi f_c t + \phi(t))$

In a phase modulated (PM) system, the phase is proportional to the message

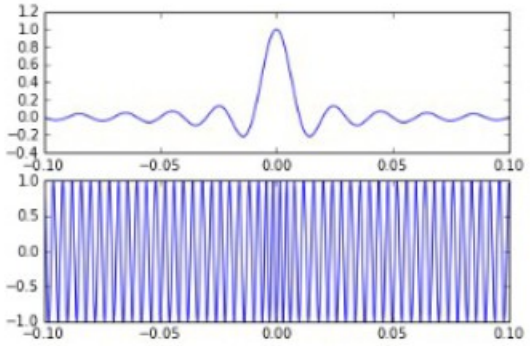
$\phi(t) = k_p m(t)$

In a frequency modulated (FM) system, instatantaneous frequency deviation is proportional to the message

$$f_i(t) - f_c = k_f m(t) = \frac{1}{2\pi}\frac{d}{dt}\phi(t)$$

In [12]:
```
from numpy.fft import fft,fftfreq
t = arange(-0.1,0.1,0.0001)
m = sinc(100*t)
int_m = empty(len(t))
for k in range(len(t)):
    int_m[k] = trapz(m[0:k],t[0:k])
u = cos(2*pi*250*t + 2*pi*100*int_m)
subplot(211)
plot(t,m)
subplot(212)
plot(t,u)
```

Out[12]: [<matplotlib.lines.Line2D at 0xd3a490c>]



JUPYTER NOTEBOOK: IMPORTARE MODULI

- consideriamo il modulo che include la definizione di due funzioni per il calcolo dei numeri di `Fibonacci`

```
# File fibo.py - Fibonacci numbers module
def fib(n): # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
        print()

def fib2(n): # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

- il modulo può essere importato nel notebook e le funzioni possono essere poi invocate

In [26]:

```
1 from fibo import fib, fib2 # or from fibo import *
2 print(fib2(500))
3 fib.__module__ # special attribute __module__
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

JUPYTER NOTEBOOK: IMPORTARE MODULI

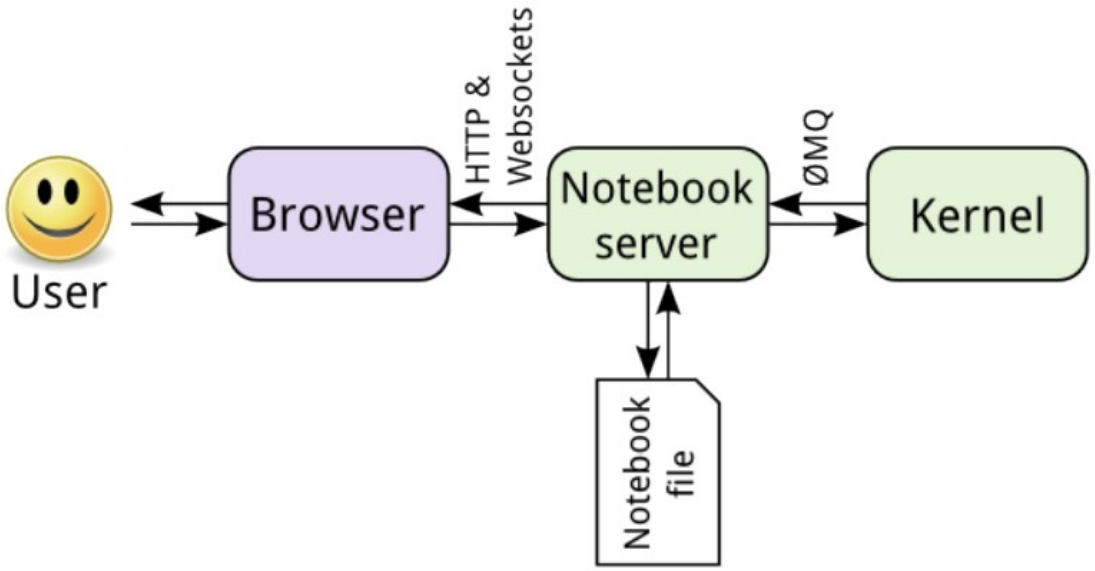
In [13]:

```
1 import fibo # imports module from local file
2 fibo.fib(6) # dot notation
3 print(fibo.__name__) # special attribute __name__
4 print(fibo.fib.__module__) # special attribute __module__
5
```

```
1
1
2
3
5
fibo
fibo
```

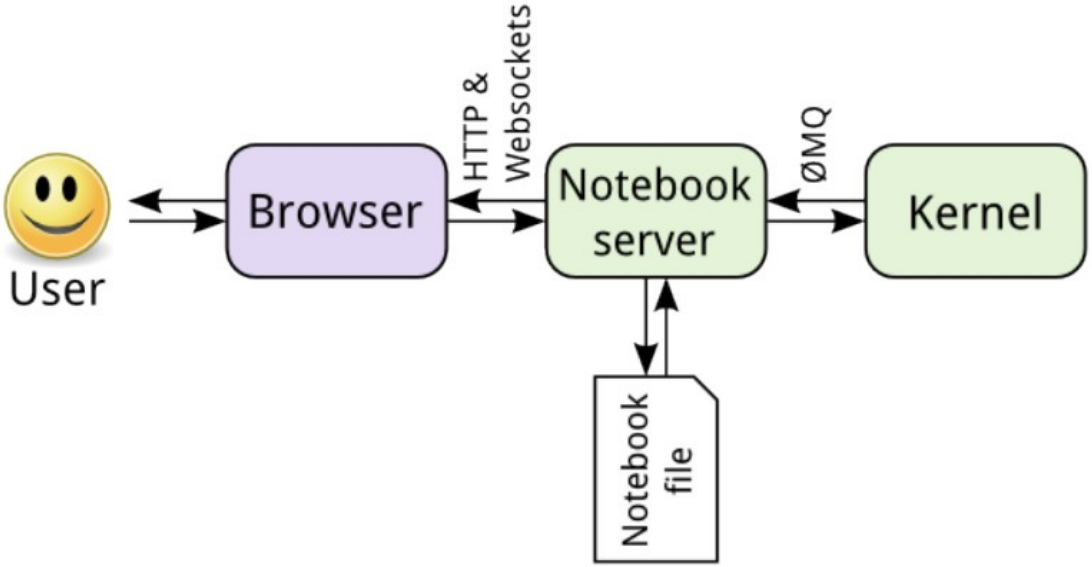
- il modulo può essere importato nel notebook e le funzioni possono essere poi invocate

JUPYTER NOTEBOOK: ARCHITETTURA



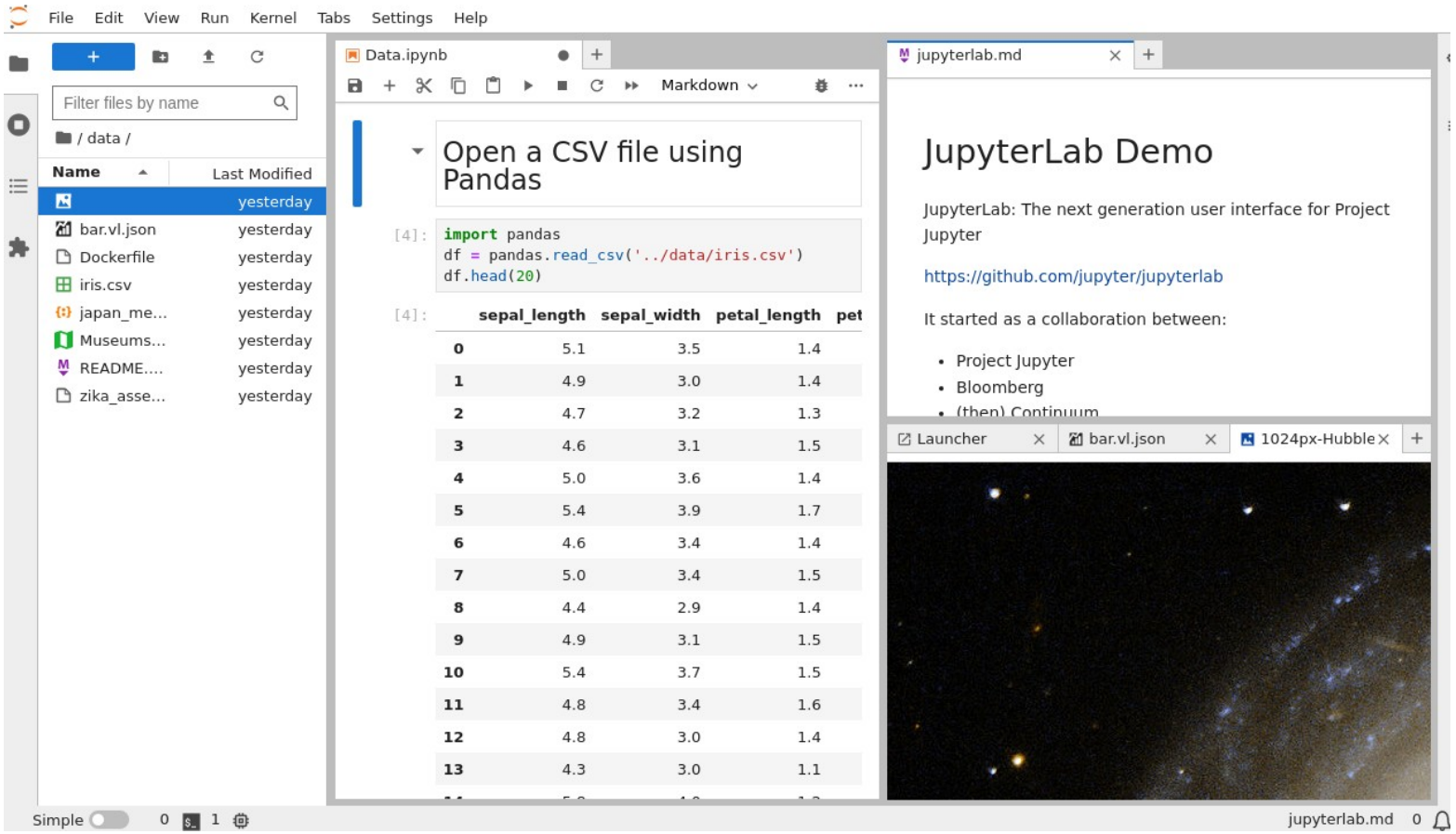
- invocabile da linea di CLI
  ```
  > jupyter Notebook
  ```
- eseguito in una finestra del browser
- interagisce con un `HTTP Notebook server` locale (in genere attivo a `http://localhost:8888`) che interagisce, a sua volta, con un `Kernel process`
- browser potrebbe interagire con `Notebook server` e `Kernel` in esecuzione su un host remoto, ma `Notebook e Kernel` devono essere colocati

JUPYTER NOTEBOOK: ARCHITETTURA



- `ZeroMQ`: protocollo di comunicazione tra server e kernel che semplifica paradigmi di pubblicazione client/server o pub/sub
- contenuti del `Notebook` memorizzati in un file `JSON` (estensione `.ipynb`)
  - memorizza contenuti del notebook compresi risultato delle esecuzioni
  - condivisione di codice e di risultati

JUPYTERLAB



- applicazione desktop
- può essere lanciata dalla GUI del sistema operativo

COSA VEDREMO NELLE PROSSIME LEZIONI

- costrutti di controllo
- strutture dati:
  - sequenze
    - tuple, Stringhe, Liste
    - meccanismo della comprehension
  - dizionari, insiemi
- higher order functions: how-to
- iteratori: how-to
- … ed altro ancora...che vedremo "by example"
  - implementazione di esercizi dal calcolo delle probabilità