

Introducción a Python

Diplomatura en Ciencia de Datos

Dr. Manuel Pulido

Departamento de Física, FaCENA, Universidad Nacional del Nordeste
Instituto de Modelado e Innovación Tecnológica, CONICET
Instituto Franco-Argentino de Estudios Climáticos, IFAECI, CONICET/CNRS
<https://pulidom.github.io/>

Dr. Diego Acosta, Ing. Sebastian Suaid

Introducción a Python

Temario de la clase 1

- Características de python.
- ipython - jupyter - shell
- Convenciones de buena programación en python
- Python como una calculadora.
- Variables. Tipos de variables
- Listas. Tuplas. Diccionarios. Conjuntos.
- Print f-strings
- Uso de librerías

Que es Python?

- ▶ Lenguaje de muy alto nivel (sintaxis comprensible y muy sencilla).
- ▶ Lenguaje interprete (No necesita de compilación).
- ▶ Lenguaje estructurado. La **tabulación** es parte de la sintaxis.
- ▶ Lenguaje orientado a objetos.

Aplicaciones

- Lenguaje utilizado en la mayoría de las aplicaciones desarrolladas por Google.
- Youtube esta hecho en python.
- Las principales librerías de IA - machine learning - aprendizaje profundo estan en python.
- Usado en las grandes compañías que estan en el estado del arte en IA (facebook, openai, deepmind, etc).

Es el lenguaje dominante para aplicaciones de machine learning .

Porque Python?

- ▶ Programas muy compactos (3-4 veces mas corto que en fortran o C).
- ▶ Programas legibles.
- ▶ Lenguaje estructurado y orientado a objetos.
- ▶ Es un lenguaje open-source (gnu).
- ▶ Muy fácil /de integrar con/integrador de/ otros lenguajes C/Fortran/Java.
- ▶ Enorme comunidad de usuarios.
- ▶ Gran disponibilidad de códigos y librerías de otros usuarios. Stackoverflow

Fanatismo de python. Comunidad pythonica. ¿Que es ser pythonico?

Zen of python: 20 Mandamientos de python

Bibliografía para el módulo

Intro python:

- ▶ Gonzalez Duque, R., 2016. Python para todos.

<http://mundogeek.net/tutorial-python/>

Libros de python para data science:

- ▶ VanderPlas, J., 2016. Python data science handbook. O'Reilly Media, Inc.
- ▶ McKinney, W., 2022. Python for data analysis. O'Reilly Media, Inc..
- ▶ Grus, J., 2019. Data science from scratch: first principles with python. O'Reilly Media.

Enfocados en python y avanzados:

- ▶ Ramalho, L., 2022. Fluent python. O'Reilly Media, Inc..
- ▶ Martelli, A., Ravenscroft, A.M., Holden, S. and McGuire, P., 2023. Python in a Nutshell. O'Reilly Media, Inc.

Como usar python

- ▶ Versiones recomendadas: python 3.8 o superior.
- ▶ Anaconda es una plataforma python completa (demasiado) para data science
- ▶ Fácil instalación de librerías individuales con: `pip` o `conda`

En una terminal shell/bash/macOS. Desde línea de comando:

```
$ python
```

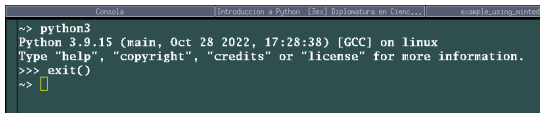
Si se quiere ejecutar en forma remota (en otra computadora):

```
$ ssh usuario@computadora
```

```
$ ssh usuario@10.40.60.207 → para loggearse en el servidor de la diplo
```

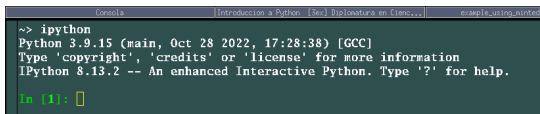
Luego en la terminal remota 10.40.60.207 se puede ejecutar el python:

`$ python` Interprete es-
tandard.



The screenshot shows a terminal window with a dark background. The title bar includes tabs for 'Console', 'Introducción a Python', '3ex) Diplomatura en Cienc...', and 'example_using_jupyter'. The terminal output shows the command '~> python3' followed by the Python 3.9.15 startup banner: 'Python 3.9.15 (main, Oct 28 2022, 17:28:38) [GCC] on linux', 'Type "help", "copyright", "credits" or "license" for more information.', and the prompt '>>>'. The user enters 'exit()' and the prompt changes to '~>'.

`$ ipython` Interprete mejo-
rado.



The screenshot shows a terminal window with a dark background. The title bar includes tabs for 'Console', 'Introducción a Python', '3ex) Diplomatura en Cienc...', and 'example_using_jupyter'. The terminal output shows the command '~> ipython' followed by the IPython 3.13.2 startup banner: 'Python 3.9.15 (main, Oct 28 2022, 17:28:38) [GCC]', 'Type "copyright", "credits" or "license" for more information', and 'IPython 3.13.2 -- An enhanced Interactive Python. Type "?" for help.'. The prompt is 'In [1]:'.

Para salir del intérprete :

```
>>> exit() o Ctrl-D
```

Python como una calculadora

Operaciones aritméticas: Suma, Resta, Multiplicacion, Division, Potencias, Raices

```
>>> 5+15
>>> 5.2/3.1 + 2
>>> 5/3**2
>>> 5+3**2
>>> (5+3)**2
>>> (5+3)**2+1
>>> (5+3)**(2+1)
```

Explicite en que orden se realiza cada una de las operaciones

Orden de las operaciones aritméticas: 1. **, 2. *, /, 3. +, -.

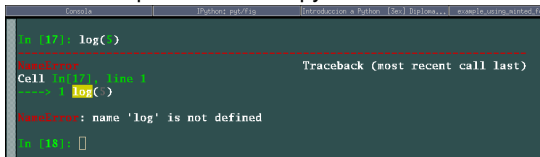
Podemos alterar el orden con ()

Estas 6 son todas las operaciones aritméticas que sabe hacer python!

Probar con

```
>>> log(5)
>>> sin(3.5)
```

Que resulta?

A screenshot of a Jupyter Notebook's console window. The window has tabs at the top: 'Console', 'IPython: put/fig', and 'Introducción a Python [Dev] Biblioteca...'. The console shows the following text:

```
In [17]: log(5)
-----
NameError                                Traceback (most recent call last)
Cell In[17], line 1
----> 1 log(5)
NameError: name 'log' is not defined

In [18]:
```

***Python no sabe de logaritmos, ni senos, ni cosenos, etc. ***

Variables en python

- ▶ Las variables en python no necesitan ser declaradas de antemano. (tipadas dinamicamente).
- ▶ Cuando se le asigna el valor a una variable se declara automaticamente:

```
>>> a = 5
```
- ▶ Una variable puede asignarse a cualquier tipo de objeto (de los del base).
- ▶ Por supuesto requiere ser asignada antes que queramos utilizarla.
- ▶ Para estructuras de datos es conveniente usar los objetos del base. Son mas eficientes.
- ▶ Fuertemente tipadas. Las operaciones válidas son las construidas en el base.

La flexibilidad viene con el costo de la eficiencia. En c/fortran una variable es una etiqueta de la posición en memoria donde se encuentra el valor numérico.

En python una variable es un puntero a un objeto que contiene información extra.

Tipo de variables

```
>>> i=5
>>> p=5/2
>>> j=5//2
>>> s='Hola'
>>> lpaso=True
>>> a=5+2j
```

Tipos de variables: Enteros. Flotantes. Cadena de caracteres. Lógicas. Números complejos.

Para conocer de que tipo es una variable:

```
>>> type(a)
```

Podemos transformar a las variables entre tipos consistentes:

```
>>> complex(5.0)    → Transforma de float o integer a complex.
>>> int(5.0)         → Transforma a integer.
>>> float(5)         → Transforma a float.
>>> str(5.0)         → Transforma a string.
```

***El float por default en python es doble precisión (c/fortran) ***

Primer programa python

Para código largos nos conviene escribirlo en un archivo y guardarlo a este lo denominamos “programa” o “script”.

Los nombres de los archivos python tienen extensión .py.

Para escribir un programa se usa cualquier editor: emacs, vi, nano y hasta el notepad de windows puede servir.

Editemos un archivo:

```
$ nano simple.py
```

```
#!/usr/bin/env python3
pi=3.141592
radio=20.0
per=2*pi*radio
sup=pi*radio**2
print ('El perimetro es: ',per)
print ('La superficie es: ',sup)
```

Ejecución de un programa python

Para ejecutar un programa python lo que hacemos en una terminal shell/bash es:

```
$ python simple.py
```

Si el programa se encuentra en otro directorio se le da el camino completo:

```
$ python /home/pulido/curso/ml/pyt/simple.py
```

Para hacer el **programa ejecutable**, hacer

```
$ chmod +x simple.py
```

luego **ejecuto directamente** el script: `$./simple.py`

Para esto es importante incluir como primera línea en el script que interprete utiliza:

```
#!/usr/bin/python3.11
```

Se puede seleccionar la version del interprete. Incluso si lo quiero interpretar con un **virtual environment**:

```
#!/home/pulido/bin/venv/python
```

Interprete: help y exit

Si queremos consultar información de un comando en el interprete:

```
>>> help(input)
```

```
>>> help(print)
```

Para salir del interprete:

```
>>> quit()
```

```
>>> exit()
```

o [Ctrl] + d

Si es en un script y quiero terminar un programa en el medio:

```
...  
raise SystemExit %Este es el recomendado en un programa.  
...
```

No se debe usar `quit()` ni `exit()`

En el caso que quiera detener una ejecución de un script:

[Ctrl] + c

Literate programming

Readability counts

Comentarios en el programa:

- ▶ Para recordar lo que hicimos (nosotros mismos).
- ▶ Para explicar lo que hacemos en el código a otro programador.
- ▶ Para referenciar el objetivo, que es lo que hace, cuando lo hicimos, cuando lo modificamos, que cosas necesitamos agregarle, etc.

REGLA DE ORO: Es esencial que todo programa este comentado hasta el último detalle.

- ▶ Los comentarios de una sola línea se hacen con **#**:
(todo lo que sigue detrás del símbolo python lo interpretará como un #comentario)
5+8 # suma
- ▶ Comentarios generales de una línea **"hola"**
- ▶ Comentarios de varias líneas se hacen con: **"""** (triple comillas)

En python las comillas y los apóstrofes (comillas simples) son equivalentes.

Comentarios para documentación de un código

Los comentarios con triple comillas que aparecen al principio de funciones, módulos, o clases de python son “docstrings”.

- ▶ Sirven para generar la documentación de todos los módulos de python.
- ▶ La docstring de una función la pueden ver en el atributo: `__doc__`.

Supongamos el script `simple.py` con una docstring:

```
""" Calcula el
    perimetro y la superficie de una circunferencia.
    MP. [2023-08-10]
    TODO. Agregar el volumen de una esfera
    """
pi = 3.141592
radio=15
perimetro = 2*pi*radio    # perimetro de la circunferencia
superficie = pi*radio**2
print ('El perimetro es: ',perimetro)
print ('La superficie es: ',superficie)
```

Probar en el interprete:

```
>>> import simple → Importa/carga el script
>>> help(simple)
```

docstrings formales con matemática

```
def siren_uniform(tensor: torch.Tensor, mode: str = 'fan_in', c: float = 6):  
    r"""Fills the input 'Tensor' with values according to the method  
    described in 'Implicit Neural Representations with Periodic Activation  
    Functions.' - Sitzmann, Martel et al. (2020), using a  
    uniform distribution. The resulting tensor will have values sampled from  
    :math:\mathcal{U}(-\text{bound}, \text{bound})' where  
    .. math::  
        \text{bound} = \sqrt{\frac{6}{\text{fan\_mode}}}
```

Also known as Siren initialization.

Examples:

```
>>> w = torch.empty(3, 5)  
>>> siren.init.siren_uniform(w, mode='fan_in', c=6)  
  
:param tensor: an n-dimensional 'torch.Tensor'  
:type tensor: torch.Tensor  
:param mode: either ''fan_in'' (default) or ''fan_out''. Choosing  
    ''fan_in'' preserves the magnitude of the variance of the weights in  
    the forward pass. Choosing ''fan_out'' preserves the magnitudes in  
    the backwards pass.  
:type mode: str, optional  
:param c: value used to compute the bound. defaults to 6  
:type c: float, optional  
"""
```

Convenciones nombres de variables

REGLA DE ORO: Las variables que definimos deben tener nombres representativos de la información que contiene e.g. `superficie` (variable que guarda la superficie).

- ▶ Por convención y visibilidad mejor minúsculas.
- ▶ Dependiendo el contexto se puede usar la raíz de la palabras siempre que quede definida la variable sin confusiones. E.g. `sup`, `perim`
- ▶ `minusculas_con_guiones` para funciones, métodos, atributos y variables
- ▶ `minusculas_con_guiones` o `TODO_MAYUSCULAS` para las constantes
- ▶ `PalabrasPrimeraEnMayusculas` para las clases

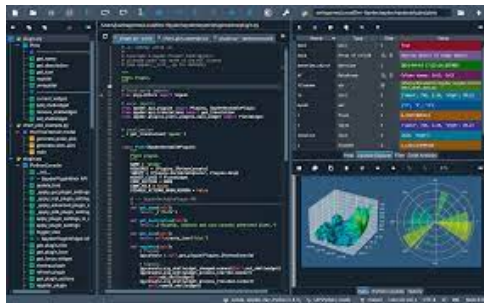
Lenguajes como el fortran identifican las variables no definidas explícitamente a través de la primera letra. Las variables que comienzan con `i`, `j`, `k`, `l`, `m`, or `n` son enteros. Podemos usar para los índices `i_x` `i_dia` o dimensiones `n_x` `n_observaciones`.

Para python existe un conjunto de **convenciones** o **normas** de estilo propias: **Python Enhancement Proposal**

Para la escritura de códigos python ver las PEP8 <https://peps.python.org/pep-0008/>

Entornos de trabajo

Integrated developed enviroment IDE: editor de textos, interprete/builder, debugger, visualización

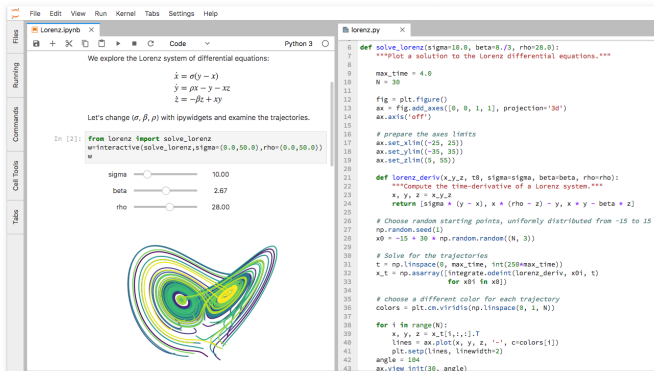


Open source: spyder.

Disponible en repositorio python / anaconda.

Proprietary: pycharm, visual studio code (vsc)

Cuadernos de trabajo



JUPYTER
JULia, PYThon, y R.

jupyter notebook: entorno web IDE con código + markdown + figuras

<https://jupyter.org/>

Como las notebooks se usan en un web browser el proceso puede correr en cualquier servidor remoto.

No sirve para producción! Solo para testing/análisis.

Alternativa nerd: emacs + org-babel (iterate programming multilanguage)

Link org-python para más detalles

Entorno de trabajo: ipython + editor de textos

ipython + editor de textos: ipython: interprete/builder interactivo y debugger

- ▶ Es una shell de comandos interactiva que interpreta python.
- ▶ Óptimo para exploración. Se miran los datos. Se evalúan nuevas ideas.
- ▶ Flujo de trabajo continuo en lugar del edit-compile-run.
- ▶ Las variables quedan en el namespace durante toda la sesión.

Entorno ideal para data science

Una vez que está todo definido si podemos construir un código estático para producción.

Opciones nerd:

emacs + flycheck + PEP8 + ipython

vim + vim-polyglot + ALE + ipython

<https://www.emacswiki.org>

Conceptos de ipython (que se extienden a jupyter)

- ▶ Comandos mágicos. Inspirados en la shell.
- ▶ [TAB] Autocompletado de comandos, funciones, variables, archivos, y un gran etc.
- ▶ a? Signo de pregunta. Introspección. Se puede usar antes o después de una variable para mostrar información del objeto.
- ▶ Acceso al SO. Podemos navegar los archivos cd/ls. Podemos interactuar shell/python.
- ▶ Ejecución de programas. Con variables cargadas para posterior evaluación, graficación debugging.



Creador Fernando Perez

`np.linalg?` Lista de comandos accesibles que empiezan de esa forma.

Comandos mágicos en ipython y jupyter

Los comandos mágicos comienzan con `%` (o sin nada).

`%run` ejecuta los scripts

`%run -i` ejecuta con lo que ya tiene cargado en el namespace

`%lsmagic` Lista todos los comandos mágicos disponibles.

`%reset` limpia el namespace

`%edit` edita un script y lo ejecuta

`%automagic` para usar los comandos mágicos sin el `%`

`%history` historia de comandos anteriores.

`%pwd` donde estoy parado (print working directory)

`%cd` cambio directorio

`%ls` list

`%export` Exporta la historia a una jupyter notebook (ipynb)

`%logstart nombre.py` Guarda la historia en un archivo.

```
Console | Introduccion a Python | Ses | Exploratura en Ciencia de Datos | Github

In [12]: %lsmagic
Out[12]:
Available line magics:
%import %alias %alias_magic %autoawait %autocall %autoindent %automagic %auto
reload %bookmark %cat %cd %clear %colors %conda %config %cp %cpaste %debug
%dhist %dirs %doctest_mode %ed %edit %env %gui %hist %history %killbgscript
%s %mkdir %less %lf %lk %ll %load %load_ext %loadpy %logoff %logon %logstart
%logstate %logstop %ls %lsmagic %lx %macro %magic %man %matplotlib %mkdir
%more %mv %notebook %page %paste %pastebin %pdb %pdef %pdoc %pfile %pinfo
%pinfo2 %pip %popd %pprint %precision %prun %psearch %psource %pushd %pwd
%pycat %pylab %quickref %recall %rehashx %reload_ext %rep %rerun %reset %res
et_selective %rm %rmdir %run %save %sc %set_env %store %sx %system %tb %ti
me %timeit %unalias %unload_ext %who %who_ls %whos %xdel %xmode

Available cell magics:
%%! %%HTML %%SVG %%bash %%capture %%debug %%file %%html %%javascript %%js %
%latex %%markdown %%perl %%prun %%python %%python2 %%python3 %%ruby %
%script %%sh %%svg %%sx %%system %%time %%timeit %%writefile

Automagic is ON, % prefix IS NOT needed for line magics.

In [13]:
```

Interacción con la shell

Si quiero ejecutar un comando de la terminal: shell, bash, zsh, dos?? lo que tengo que hacer es comenzarlo con !

Si hago: `>>> !printenv`

Me imprime las variables del entorno/environment. Otro el echo:

```
>>> !echo $BASH_VERSION
4.4.23(1)-release
```

Para comandos mas complejos conviene utilizar **cell magics**: `%%`

La celda esta pensada para múltiples comandos/líneas. Es un concepto muy usado en el jupyter.

```
>>> %%bash
. .../.bashrc
<comandos_bash> <argumentos>
```

Puedo escribir y guardar un python script desde ipython:

```
>>> %%writefile foo.py
print('Hello world')
```

Para ejecutar scripts en cualquier interprete, python, Ruby, R, etc:

```
>>> >>> %%script python3.11
```

Puedo mezclar python con bash/terminal, mando una variable python a la shell:

Debugger interactivo con ipython

Que hago cuando aparecen un problema y no se donde esta?

Python tiene un debugger incorporado: dbg.

Comandos esenciales:

n next ejecuta la línea

p a,b,c imprime variables

q si queremos salir sin seguir ejecutando.

c si queremos continuar la ejecución sin seguir debugging

```
import pdb
...
# donde se quiere que comience
# a rastrear el bug agregar
pdb.set_trace()
```

- ▶ Dentro de ipython se puede invocar el debugger con %debug
- ▶ Se puede usar **después que aparezca la excepción!**
- ▶ Si nos queremos anticipar podemos setear el debugger: %pdb
- ▶ Otra forma de invocarlo: %run -d -b40 script.py -d debugger, -b breakpoint en linea 40

Opción mas artesanal: Se pueden introducir **print** de las variables antes de la línea donde aparece el error para ver lo que esta sucediendo.

Profiling en python

Para medir el **tiempo de cómputo** de un comando o una función en ipython:

```
%time fn()  
%timeit fn()
```

`timeit` hace múltiples evaluaciones para una mas robusta medida del tiempo de comandos cortos (media y desviación).

Si se quiere evaluar el tiempo que se tarda todo un script y que funciones demandan mas tiempo/cómputo estan los **perfiladores**:

```
$ python -m cProfile cprof_example.py  
$ python -m cProfile -s cumulative cprof_example.py
```

Dentro de ipython el profiler:

```
In [1] %prun -l 7 -s cumulative run_experiment()  
In [2] %run -p -s cumulative cprof_example.py
```

→ ordena por tiempos acumulados

Para medir la **memoria** utilizada:

```
%memit fn()  
%mprun main.py
```

→ memoria que requiere un comando.

→ perfilador de memoria (MemoryProfiler)

Print con formato de salida

Convierte una expresión a cadena de caracteres y muestra la cadena en la terminal. En python 3 se considera una función con multiples argumentos, es decir se usa con parentésis:

```
print ('El monto es: ',a)
```

La coma agrega un espacio en blanco. Puedo usar `\n` para salto de líneas. Supongamos que tenemos un examen y queremos informar los estudiantes presentes

```
print ('La cantidad de asistentes al examen fue de %d alumnos de un curso de \n
      %d alumnos. Los ausentes fueron %d' \n
      (nasis,ncurso,ncurso-nasis))
```

Las notas se deben informar con uno o dos dígitos pero el promedio con dos decimales.

```
print ('Perez, Juan      %3d'   %(nota1))
print ('Sanchez, Pedro %3d'   %(nota2))
print ('Promedio        %6.2f'  %(0.5*(nota1+nota2)) )
```

%d se utiliza para enteros. De utilidad para columnas.

%f para flotantes. `%(digitos).(decimales)f`

%s para una cadena (string).

f-strings

Una forma nueva (python >=3.6) y mucho mas conveniente de imprimir variables con texto son las f-strings:

```
print (f'La cantidad de asistentes al examen fue de {nasis} alumnos.')
```

```
print (f'de un curso de {ncurso} alumnos.')
```

```
print (f'Los ausentes fueron {ncurso-nasis}')
```

Con formato:

```
print (f'Perez, Juan   {notal:d}')
```

```
print (f'Sanchez, Pedro {nota2:d}')
```

```
print (f'Promedio   {(0.5*(notal+nota2)):6.2f}' )
```

Conjuntos de variables: listas

Una **lista** es un **conjunto** de variables de cualquier tipo separados por comas y delimitado por corchetes. La creo:

```
lista=[25,60.4,'edad y peso','domicilio']
```

- ▶ Para acceder a un elemento de la lista:

```
>>> print ( lista[1] )
```
- ▶ Para acceder a varios elementos de la lista:

```
>>> print ( lista[1:3] )
```
- ▶ Cantidad de elementos de la lista:

```
>>> print ( 'Longitud: ',len(lista))
```
- ▶ Si quiero cambiar la edad en la lista:

```
>>> lista[0]=26
```

Conjuntos de variables: tuplas

Las tuplas son secuencias de objetos como las listas pero no se pueden cambiar (no mutables) `>>> tupla=(25,60.4,'edad y peso')` si engordo y quiero cambiar el peso:

```
>>> tupla[1]=61.6
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

La razón de su existencia es que son mucho mas eficientes que las listas. Son utilizadas como argumentos de entrada y salida de las funciones como ya veremos en las próximas clases.

Objetos por todos lados

Las variables son objetos en python

```
a=15.0  
dir(a) # metodos disponibles para los flotantes  
a.is_integer()
```

Las cadenas de caracteres, las listas y las tuplas son **secuencias** (y objetos).

- ▶ Las secuencias son iterables.
- ▶ Las operaciones sobre las secuencias se aplican a todas por igual. Incluso a otros containers (e.g. diccionarios)

Operaciones con listas y strings

Hay un conjunto de funciones “built-in” (nativas) que son aplicables a las secuencias: listas, tuplas y strings:

```
len(seq), max(seq), min(seq), sum(seq)
```

donde `seq` una secuencia (e.g. lista).

Mapa:

`map(funcion, lista)`: aplica la función `funcion` a todos los elementos de la lista `lista`

```
>>> lista = ['Ricardo', 'RODOLFO', 'sergio' ]
```

```
>>> list(map(str.lower, lista))
```

```
['ricardo', 'rodolfo', 'sergio']
```

Operaciones con listas

`range(<inicio,>fin<,salto>)`: crea una lista de enteros, desde 0 de uno en uno (el fin esta excluido!).

```
>>> print ( range(4) )  
[0, 1, 2, 3]  
>>> print ( range(2,10,2)  
[2, 4, 6, 8]
```

Rebanadas - slicing

Var [] Corchetes aplican a secuencias

Var [<inicio,>fin<,salto>]

```
a=list(range(10))
```

Las secuencias en python3 tienen “lazy evaluation”

Formas posibles:

```
a[2:5]  
a[2:]  
a[:5]  
a[:-2]  
a[::-2]
```

Y si hacemos: `a[::-2]` ?

Operaciones con listas

```
>>> z = [ ]
```

 Crea una lista

lista.append(elemento) Agrega elementos a una lista

```
>>> z = [1, 2.02]
>>> z.append(800.8)
>>> z
[1, 2.02, 800.8]
```

lista.reverse(): invierte la lista

```
>>> z.reverse()
>>> print ( z )
[800.8, 2.02, 1]
```

lista.remove(x): elimina el primer elemento que coincide con x de la lista

lista.pop(j): elimina el elemento j-esimo de la lista

Operaciones con cadenas de caracteres

Subcadenas `[i:j]` (slicing de strings):

```
>>> sa='cadena'  
>>> print ( sa[2:4] )  
de
```

Transforma un número en cadena, `str`

```
>>> a=1239  
>>> b=str(a)  
>>> print ( b[2:4] )  
39
```

Concatena una cadena: `+`

```
>>> nombre='Juan'  
>>> apellido='Perez'  
>>> nombre_completo=nombre+' '+apellido  
>>> print ( nombre_completo )  
Juan Perez
```

¿Qué produce con `*` en las strings? e.g. `a=5*'z'`

Operaciones con cadenas de caracteres

Cambia a mayúsculas: `.upper()`

```
>>> a='casa'
>>> print ( a.upper() )
CASA
```

Reemplaza un caracter o cadena de caracteres: `.replace(viejo,nuevo)`

```
>>> print ( a.replace('a','o') )
coso
```

Busca un caracter o cadena de caracteres: `.find('o')`

```
>>> a='casona'
>>> print ( a.find('o') )
3
```

Existen mas de 30 métodos para las cadenas, ver con `>>> dir(a)`

Conjuntos de variables: Diccionarios

Almacenan un conjunto de variables u objetos.

- ▶ Se accede a sus elementos a partir de claves principales (keys).
- ▶ La clave sirve para identificar al elemento.
- ▶ Se definen con llaves (en lugar de corchetes que es para las listas).

```
>>> x={'nom','dom'}
```

Asignación:

```
>>> x={'nom':'Sergio','dom':'Libertad 5400'}  
>>> x['nom']='Sergio'; x['dom']='Libertad 5400'
```

Funciones para diccionarios: `len()`, `x.keys()`, `x.values()`

Mas info del diccionario: `>>> a={}; dir(a)`

Información sobre un método: `>>> help(a.popitem)`

Uso de librerías

```
>>> import math
>>> math.log(5.0)
```

Si queremos renombrar (por ejemplo para que el nombre sea mas corto):

```
>>> import math as m
>>> m.log(5.0)
```

Si queremos importar solo un código que esta dentro de una librería:

```
>>> from math import log, sin
>>> log(5.0)
```

Notar que con esto reconoce el comando directamente. No es aconsejable en programación pero si para hacer pruebas.

Si la libreria tiene estructura jerarquica, libreria adentro de otra (estructura de directorios):

```
>>> import numpy.random as rnd
```

Prohibido: `from math import *`

Librerías específicas

- ▶ `math`, `cmath` Funciones matemáticas
- ▶ `sys` Parametros del sistema
- ▶ `os` Variables y comandos de la shell.
- ▶ `numpy` Matrices y vectores. Operaciones. Wrappers to Blas, Lapack, fft.
- ▶ `numpy.random` Generador de números aleatorios.
- ▶ `matplotlib` Graficación.
- ▶ `tkinter`. Interface a Tcl/Tk Interfaces gráficas para int. usuario.
- ▶ `threading` Paralelismo por hilos.
- ▶ `multiprocessing` Paralelismo por procesos.
- ▶ `pandas` Librerías de manipulación de datos
- ▶ `hdf5`, `netcdf` Lectura y escritura de datos de alta dimensionalidad.
- ▶ `scipy`, `sklearn`, etc aprendizaje automatizado
- ▶ `jax`, `pythorch`, `tensorflow`, `keras` redes profundas.

Uso de librerías. os

Si queremos ejecutar algun comando de terminal shell o bash, esta la librería **os**.

```
>>> import os
>>> os.system('ls')
```

- Si queremos chequear si existe un archivo: **isfile**

```
>>> import os
>>> os.path.isfile('prueba.tex')
```

- Si queremos chequear si existe un directorio o un archivo: **exists**

```
>>> os.path.exists('prueba.tex')
```

- Si queremos chequear si existe un directorio: **isdir**

```
>>> os.path.isdir('/home/programacion/readme')
```

Librería **math**. Funciones matemáticas

La librería `math` posee todas las funciones matemáticas standards: `log`, trigonométricas: `sin`, `cos`, `tan`, función error, factorial, etc.

Además posee constantes matemáticas: e , π .

De pasaje de radianes a grado: `>>> math.degrees(2.*math.pi)`

Entonces con el python solol tenemos una calculadora de las truchas, con el python y el `math`, tenemos una calculadora científica, y con el `scikit-learn` ...

Uso de librerías: datetime

Para manejo de fechas, python tiene la librería **datetime**

Si se necesita conocer la fecha de hoy:

```
>>> import datetime
>>> datetime.date.today()
```

Si se necesita introducir una fecha:

```
>>> date1=datetime.datetime(2007,1,1)
```

Para pasar una fecha a número de días: **date.toordinal()**

Para pasar un número de días a fechas: **datetime.date.fromordinal(4535)**