PythonBasicCourse-es002

October 10, 2022

1 Curso básico de Python

1.1 Apuntes

Curso básico de Python. Apuntes por Marcelo Horacio Fortino. Versión 2.3. Octubre 2022.

Versión	Autor/es	Fecha	Observaciones
1.0	Marcelo Horacio Fortino	2021/Marzo	Curso Python
1.1	Marcelo Horacio Fortino	2021/Junio	Convertido a markdown - ipynb
1.2	Marcelo Horacio Fortino	2021/Agosto	Actualizados contenidos
1.3	Marcelo Horacio Fortino	2021/Octubre	Agregado Flask microframework
1.4	Marcelo Horacio Fortino	2021/Noviembre	Agregado Pandas - datascience
1.5	Marcelo Horacio Fortino	2021/Diciembre	Agregado Devops - Ansible
2.0	Marcelo Horacio Fortino	2022/Abril	Nueva estructura: core / module
2.1	Marcelo Horacio Fortino	2022/Junio	Módulo apuntes intermedio
2.2	Marcelo Horacio Fortino	2022/Agosto	Actualizado temario y ejercicios
2.3	Marcelo Horacio Fortino	2022/Octubre	Actualizado despliegue a render.com

cluso sin la garantía MERCANTIL implícita o sin garantizar la CONVENIENCIA PARA UN PROPÓSITO PARTICULAR. El autor no asume ninguna responsabilidad si el lector hace un mal uso de la misma.

- Estos apuntes se basan en:
  - La documentación oficial de Python, https://docs.python.org/es/3/tutorial/index.html,
  - La bibliografía presentada al final de este documento, y
  - Documentación propia recogida a lo largo de los años de diversas fuentes.

1.2 Objetivo del curso

- Objetivo general del curso:
- Aprender a usar Python para crear scripts y programas simples plenamente funcionales, desarrollar aplicaciones web y realizar análisis de datos.

- Objetivos específicos:

- Reconocer las características principales de Python y su utilidad práctica.
- Identificar tipos de datos (simples y compuestos) y operadores.
- Aplicar variables y estructuras de control de flujo.
- Construir funciones y clases (POO).
- Clasificar los módulos y paquetes por sus funcionalidades y objetivos.
- Comparar el lenguaje con otros similares de scripts, procedimentales y orientados a objetos.
- Probar bibliotecas para conexiones REST a aplicaciones web y bases de datos.
- Resolver problemas y errores en el código fuente proponiendo soluciones alternativas (refactoring).
- Utilizar la biblioteca pandas junto con matplotlib y numpy para realizar análisis estadísticos de datos y gráficos.

- Como resultado práctico al final del curso cada estudiante habrá creado una aplicación web utilizando el microframework de Python Flask.

1.3 Temario

- Introducción, instalación y compilación
- Datos, expresiones y sentencias
- Variables y funciones, control de flujo
- Clases y objetos, herencia, polimorfismo
- Entradas y salidas con Python
- Gestión de módulos, paquetes y bibliotecas
- Servicios y programas en red, REST API
- Desarrollo de aplicaciones web con Flask
- Análisis de datos con pandas, matplotlib y numpy

- Módulos opcionales
  - SQL ejemplos con pandas
  - Plotting con Python
  - Machine Learning con Python
  - DevOps con Ansible
  - SDK de GCP para Python

- Kubernetes con Python y Docker

1.4 Bibliografía

- Downey, A., Elkner, J., Meyers, C. Aprenda a Pensar Como un Programador con Python. (2015).
  Recuperado de https://argentinaenpython.com/quiero-aprender-python/aprenda-a-pensar-como-un-programador-con-python.pdf
- Kent D. Lee. Python, Programming Fundamentals Second Edition. 2014.
- Marzal Varó, A., Gracia Luengo, I., García Sevilla, Pedro. Introducción a la programación con Python 3. (2014).
  Recuperado de http://repositori.uji.es/xmlui/handle/10234/102653
- Miller, B., Ranum, D. Solución de problemas con algoritmos y estructuras de datos usando Python. Traducido por Mauricio Orozco-Alzate, Universidad Nacional de Colombia - Sede Manizales.
  Recuperado de https://runestone.academy/ns/books/published/pythoned/index.html?mode=browsing
- Shaw, Z. A., Learn Python 3 the Hard Way. (2016).
  Recuperado de https://learnpythonthehardway.org/
- Van Rossum, G. and the Python development team. Documentación de Python en español. (2020).
  Recuperado de https://python-docs-es.readthedocs.io/_/downloads/es/pdf/pdf/

2 Datos, expresiones y sentencias

- Objetos en Python, Tipos de datos en Python, Operadores aritméticos, Prioridad de los operadores, Conversión de tipos de datos, Datos mutables e inmutables, Datos especializados y datos integrados, Datos como listas, Tuplas en Python, Uso de diccionarios, Operadores lógicos con números, Cadenas alfanuméricas, Acceso a los elementos de una cadena, Subcadenas, Operaciones con cadenas, Funciones de cadenas, Ipython, IPython comandos útiles, Magic functions en Ipython, Alias en IPython, Jupiter Notebook.

2.1 Objetos en Python

- En Python todo es tratado como un objeto.
- Todo objeto tiene tres atributos: identidad, tipo y valor.
- Mientras que la identidad y el tipo no pueden ser cambiados una vez creados, los valores de objetos mutables sí pueden.

- Identidad: dirección del objeto en la memoria.
- Tipo: entero, lista, cadena, etc.
- Valor: valores almacenados por el objeto.

[ ]:
```
tipos = "Hola mundo!"
print("Identidad:", id(tipos))
```

```
print("Tipo:", type(tipos))
print("Valor:", tipos)
```

2.2 Tipos de datos en Python

- Los números (*numbers*), catalogados como datos simples, son de tipo entero (-5, 0, 5, etc.) o de de tipo real, llamados también punto-flotantes, o simplemente flotantes (por ej. 3,14).
- Las cadenas (*strings / str*) son datos estructurados: caracteres alfanuméricos que representan texto y se escriben entre comillas simples o dobles (por ej. "Hola Mundo!").
- Para verificar la clase a la que pertenece un dato se utiliza el comando *type()*:

[ ]:
```
type(1)
```

[ ]:
```
type(2.3)
```

[ ]:
```
type("Hola Mundo!")
```

- Literales booleanos:
- Los datos de tipo booleano (*boolean*) utilizan los elementos *True* y *False* para los valores lógicos verdadero y falso, denotados como 1 y 0.

[ ]:
```
type(True)
```

- El Sistema Binario es un sistema numérico que emplea 2 como su base: 0 y 1.
  - 0010 representa al número 2 en decimal.
- Si un número entero esta precedido por un código `0O` o `0o` (cero- letra o), el número será tratado como un valor octal.
  - 0o123 es un número octal con un valor (decimal) igual a 83.
- Si un número entero esta precedido por un código `0x` (cero- letra x), el número será tratado como un valor hexadecimal.
  - 0x123 es un número hexadecimal con un valor (decimal) igual a 291.
  - https://es.wikipedia.org/wiki/Sistema_hexadecimal.
- La notación `[0-9].E[0-9]` (también la letra minúscula `e`) significa por diez a la n potencia.
- La base puede ser un valor entero o real (el valor antes de la `E`), el exponente (el valor después de la `E`) debe ser un valor entero.

[ ]:
```
# Ejemplo constante de Planck (denotada como h)
# Su valor es: 6.62607 x 10-34.

6.62607015e-34
```

- Para trabajar en Python con constantes se utiliza la biblioteca scipy.
- https://docs.scipy.org/doc/scipy/getting_started.html.
- Se suele utilizar para trabajar en proyectos de *Data Science, Deep Learning,* y *Machine Learning.*
- El módulo contiene constantes de matemáticas, física y la base de datos CODATA 2018.

[ ]:
```
pip install scipy
```

[ ]:
```
import scipy
print("Golden ratio:", scipy.constants.golden_ratio)
print("Constante de Planck:", scipy.constants.Planck)
```

- Existe un literal especial en Python: None.
- Es llamado un objeto de *NoneType* (ningún tipo).
- Representa la ausencia de un valor.

2.3 Operadores aritméticos

- Python cuenta con operadores para representar cómputos como la adición o multiplicación.
- Los operadores básicos son `+` (suma), `-` (resta), `*` (producto), `/` (cociente).
- Otros operadores son `//` (cociente división entera), `%` (resto división entera), `**` (potencia).

- Cuando los argumentos son enteros, el resultado es entero.
- Cuando al menos un argumento es flotante, el resultado es flotante.
- El resultado producido por el operador de división siempre es flotante.

[ ]:
```
5 * -2
```

[ ]:
```
13 / 5
```

[ ]:
```
13 // 5
```

[ ]:
```
13 % 5
```

[ ]:
```
(2 + 3) ** 2
```

- Un operador unario es un operador con solo un operando, por ejemplo, -5, o +5.
- Un operador binario es un operador con dos operados, por ejemplo, 2 + 3, o 11 % 5.

[ ]:
```
2 + 3
```

2.4 Prioridad de los operadores

- El orden de prioridad de evaluación comienza con funciones predefinidas y luego en ese orden:
  - potencias, productos y cocientes, finalizando con sumas y restas.
- Se usan paréntesis para saltar este orden.
- Operadores con la misma precedencia son evaluados de izquierda a derecha salvo la potenciación.

- *A menos que la sintaxis se proporcione explícitamente, los operadores son binarios. Los operadores en el mismo cuadro se agrupan de izquierda a derecha (excepto para la exponenciación, que se agrupa de derecha a izquierda).*
- Fuente: https://docs.python.org/es/3/reference/expressions.html#operator-precedence

Operador	Descripción
(expressions…), `[expressions...]`, {key: value…}, {expressions…}	Expresión de enlace o entre paréntesis, despliegues de lista, diccionario y conjunto
x`[index]`, x`[index:index]`, x(arguments…), x.attribute	Subscripción, segmentación, invocación, referencia de atributo
await x	Expresión await
`**`	Exponenciación
`+x, -x, ~x`	NOT positivo, negativo, bit a bit
`*, @, /, //, %`	Multiplicación, multiplicación de matrices, división, división de redondeo, resto

Operador	Descripción
`+, -`	Adición y sustracción
`<<, >>`	Desplazamientos
`&`	AND bit a bit
`^`	XOR bit a bit
`|`	OR bit a bit
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparaciones, incluyendo comprobaciones de membresía y de identidad

Operador	Descripción
not x	Booleano NOT
and	Booleano AND
or	Booleano OR
if – else	Expresión condicional
lambda	Expresión lambda
`:=`	Expresión de asignación

2.5 Conversión de tipos de datos

- Convertir tipos de datos (*type casting*) en Python se puede realizar de dos formas: implícita y explícita.
- Conversión implícita: Python la realiza automáticamente.

[ ]:
```
x = 5
y = 10.22

x = x + y
print(x)
type(x)
```

- Conversión explícita: Se usan las funciones que proporciona Python, entre ellas:*float(), str(), int(), list(), set().*

[ ]:
```
# Convertir flotante (float) a entero (integer)
x = 5.5
x = int(x)
print(x)
```

[ ]:
```
# Convertir flotante a cadena (string)
x = 5.5
print(type(x))
x = str(x)
print(type(x))
```

2.6 Datos mutables e inmutables

- En Python, existen dos tipos de datos:
  - Los mutables (*mutable*) son objetos que pueden cambiar de valor, por ej. aquellos que almacenan una colección de datos.
- Objetos mutables:
  - Listas
  - Sets
  - Diccionarios
  - Clases definidas por el usuario
- Los inmutables (*immutable*) por su parte no permiten ningún cambio.
- Objetos inmutables:
  - Números (*Integer, Rational, Float, Decimal, Complex & Booleans*)
  - Cadenas (Strings)
  - Tuplas
  - *Frozen Sets*
  - Clases definidas por el usuario

2.7 Datos especializados y datos integrados

- Python tiene varios tipos de datos especializados como:
  - fechas y horas,
  - matrices de tipo fijo (*fixed-type arrays*),
  - colas de montículos (*heap queues*),
  - colas de doble extremo (*double-ended queues*), y
  - enumeraciones.
- También proporciona datos integrados:
  - Diccionario - *dict*,
  - Lista - *list*,
  - *set* y *frozenset*; y
  - Tupla - *tuple*.
- La clase *str* se utiliza para contener cadenas de caracteres *Unicode*.
- Las clases *bytes* y *bytearray* se utilizan para contener datos binarios.
- Fuente: https://docs.python.org/es/3/library/datatypes.html.

- En cuanto a los llamados datos estructurados compuestos (*containers*) utilizados para agrupar valores se utilizan:
- listas (*lists*): secuencias ordenadas de objetos de uno o de distintos tipos.

```
[ ]: type([5, 'seis', [7, 8], True])
```

- tuplas (*tuples*): se diferencian de las listas porque son inmutables, por ej. (5, 'siete', 9).

```
[ ]: type((5, 'siete', 9))
```

- diccionarios (*dictionaries*): objetos con una clave asociada; por ej. {'pi':3.1415, 'e':2.718}.

```
[ ]: type({'pi':3.1415, 'e':2.718})
```

- sets: los elementos de un *set* deben ser únicos y no mantienen el orden cuando son declarados.

```
[ ]: un_set = set([22, 7, 6, 5, 5, 9])
     print(un_set)
     type(un_set)
```

- frozenset: similares a los sets pero son inmutables.

```
[ ]: un_frozenset = frozenset([1, 2, 3])
     print(un_frozenset)
     type(un_frozenset)
```

2.8 Datos como listas

- Las listas (*lists*) son secuencias ordenadas de objetos de uno o de distintos tipos.
- Para definir una lista se utilizan corchetes `[]` (*brackets*) y para separar los elementos la coma `,` (*comma*).
- Como convención se usan nombres plurales para las listas, aunque no es obligatorio.

```
[ ]: frutas = ['manzana', 'banana', 'pera']
```

- Se puede escribir una lista también de la siguiente manera:

```
[ ]: lista_numeros = list('12345')
```

- Para acceder a los elementos de la lista se utiliza el índice (*index*) que sigue el orden en que están enumerados.
- Su primer elemento tiene el índice 0. Se puede acceder con números negativos al final de la lista.

```
[ ]: frutas[0]
     fruta1 = frutas[0]
     ultima_fruta = frutas[-1]
     print(fruta1)
```

- Para modificar un elemento de la lista se utiliza su índice:

[ ]:
```
frutas[0] = 'naranja'
```

- Para conocer el índice de un elemento:

[ ]:
```
index = frutas.index('pera')
print(index)
```

- Agregar un elemento a la lista se puede hacer al final o en cualquier lugar de la misma:

[ ]:
```
frutas.append('piña')
frutas.insert(1, 'manzana')
```

- Agregar varios ítems a la lista:

[ ]:
```
frutas.extend(['manzana', 'piña'])
mas_frutas = frutas + ['melocoton', 'kiwi']
print(mas_frutas)
```

- Para eliminar un elemento se utiliza su índice o valor:

[ ]:
```
del frutas[-1]
frutas.remove('banana')
```

- Para eliminar un elemento de la lista se puede utilizar *pop*.
- Si no se especifica la posición del elemento, eliminará de forma predeterminada el último (-1):

[ ]:
```
ultima_fruta = frutas.pop()
primera_fruta = frutas.pop(0)
```

- Para ordenar los elementos de una lista se pueden utilizar *sort* o *sorted*:
- La función *sorted* devuelve una copia de la lista, dejando el original intacto:

[ ]:
```
print(sorted(frutas))
print(sorted(frutas, reverse=True))
```

- Con el método *sort* se puede cambiar el orden de la lista:

[ ]:
```
frutas.sort()
print(frutas)
```

[ ]:
```
frutas.sort(reverse=True)
frutas.reverse()
```

- Para conocer el tamaño de la lista se utiliza len:

[ ]:
```
cantidad_frutas = len(frutas)
```

- Para dividir una lista en una sección (*slice*):

[ ]:
```
primeras_frutas = frutas[:3]
medio_frutas = frutas[1:3]
ultimas_frutas = frutas[-3:]
```

- Para hacer una copiar de una lista:

[ ]:
```
copia_frutas = frutas [:]
```

- Para iterar sobre una lista:

[ ]:
```
for fruta in frutas:
    print(fruta)
```

[ ]:
```
for fruta in frutas:
    print(f"Nombre de la fruta: {fruta}")
```

2.9 Tuplas en Python

- La tupla (*tuple*) es idéntica a la lista salvo que no se pueden modificar los valores una vez que están definidos.
- Para definir una tupla se utilizan paréntesis `()`.
- Se pueden llegar a sobrescribir todos los elementos pero no se pueden cambiar a nivel individual.

[5]:
```
coordenadas_vigo = ("08°43'21.5", "N42°13'58.15")
for coordenada in coordenadas_vigo:
    print(coordenada)
```

```
08°43'21.5
N42°13'58.15
```

- Para sobrescribir una tupla:

[6]:
```
coordenadas_vigo = ("-8.7226400", "42.2328200")
print(coordenadas_vigo)
```

```
('-8.7226400', '42.2328200')
```

- Para borrar una tupla:

[ ]:
```
del coordenadas_vigo
```

2.10 Uso de diccionarios

- Los diccionarios en Python permiten guardar datos como pares de valores denominados llave y valor.
- Al fornecer una llave Python devuelve el valor asociado con la misma.
- Para definir un diccionario se utilizan llaves `{}` (*curly braces*).
- Los dos puntos conectan la llave y el valor; y la coma separa los pares de valores.

[ ]:
```
frutas = {'manzana':'roja', 'banana':'amarilla', 'naranja':'naranja'}
```

- Para obtener el valor asociado a una llave:

[ ]:
```
print(frutas['manzana'])
```

- Para obtener el valor usando el método *get* (devuelve el valor *None* si la llave no existe):

[ ]:
```
color_frutas = frutas.get('banana')
print(color_frutas)
```

[ ]:
```
# Los métodos disponibles son:
dict.
clear()      get()        mro()        setdefault()
copy()       items()      pop()        update()
fromkeys()   keys()       popitem()    values()
```

- Para modificar los valores en un diccionario (los agrega si no existen):

[ ]:
```
frutas['manzana'] = 'verde'
print(frutas)
print(frutas['manzana'])
```

- Para eliminar pares de valores:

[ ]:
```
del frutas['naranja']
print(naranja)
```

- Para iterar en diccionarios:

[ ]:
```
for llave, valor in frutas.items():
    print(llave, valor)
```

2.11 Operadores lógicos con números

- Devuelven un valor lógico o booleano. Operadores lógicos:

- {==} (igual que),
- {> } (mayor que),
- {< } (menor que),
- {>=} (mayor o igual que),
- {<=} (menor o igual que),
- {!=} (distinto de).

[ ]:
```
3 == 3
```

[ ]:
```
3.1 <= 3
```

[ ]:
```
-1 != 1
```

2.12 Cadenas alfanuméricas

- Las cadenas (clase *str*) son una secuencia de caracteres alfanuméricos que representan texto.
- Se escriben entre comillas sencillas o dobles.

[ ]:
```
'Python'
```

[ ]:
```
"123"
```

[ ]:
```
'True'
```

[ ]:
```
# Cadena vacía
''
```

[ ]:
```
# Cadena con un espacio en blanco
' '
```

[ ]:
```
# Inserta una nueva línea
'\n'
```

[ ]:
```
# Indentado o tabulador
'\t'
```

2.13 Acceso a los elementos de una cadena

- Cada carácter tiene asociado un índice que permite acceder a él (*indexing*).

P	y	t	h	o	n
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

[ ]:
```
'Python'[0]
```

[ ]:
```
'Python'[1]
```

[ ]:
```
'Python'[-1]
```

[1]:
```
# Índice demasiado grande
'Python'[6]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_6624/2136356153.py in <module>
      1 # Índice demasiado grande
----> 2 'Python'[6]
```

```
IndexError: string index out of range
```

2.14 Subcadenas

- Mientras que *indexing* es utilizado para obtener caracteres individuales, *slicing* permite obtener subcadenas:

[ ]:
```
'Python'[1:4]
```

[ ]:
```
'Python'[1:1]
```

[ ]:
```
'Python'[2:]
```

[ ]:
```
'Python'[:-2]
```

[ ]:
```
'Python'[:]
```

[ ]:
```
'Python'[0:6:2]
```

2.15 Operaciones con cadenas

- cadena1 + cadena2 : Devuelve la cadena resultado de concatenar las cadenas `cadena1` y `cadena2`.
- cadena1 * n : Devuelve la cadena resultado de concatenar n copias de la cadena `cadena1`.

[ ]:
```
'Me gusta ' + 'Python'
```

[ ]:
```
'Python ' * 3
```

- cadena1 in cadena2 : Devuelve True si `cadena1` es una cadena contenida en `cadena2` y False en caso contrario.
- cadena1 not in cadena2 : Devuelve True si `cadena1` es una cadena no contenida en `cadena2` y False en caso contrario.

[ ]:
```
'y' in 'Python'
```

[ ]:
```
'to' in 'Python'
```

[ ]:
```
'to' not in 'Python'
```

- También se pueden realizar operaciones de comparación de cadenas utilizando los operadores lógicos mayor, menor, igual, etc.
- Por ej: cadena1 == cadena2 devuelve True si la cadena `cadena1` es igual que la cadena `cadena2` y False en caso contrario.
- Python utiliza el orden establecido en el código ASCII.
- https://es.wikipedia.org/wiki/ASCII.

[ ]:
```
'Python' == 'python'
```

[ ]:
```
'Python' < 'python'
```

[ ]:
```
'a' > 'Z'
```

[ ]:
```
'A' >= 'Z'
```

[ ]:
```
'' < 'Python'
```

2.16 Funciones de cadenas

- len(cadena): Devuelve el número de caracteres de la cadena `cadena`.
- min(cadena): Devuelve el carácter menor de la cadena `cadena`.
- max(cadena): Devuelve el carácter mayor de la cadena `cadena`.

[ ]:
```
len('Python')
```

[ ]:
```
min('Python')
```

[ ]:
```
max('Python')
```

- cadena.upper(): Devuelve la cadena con los mismos caracteres que la cadena `cadena` pero en mayúsculas.
- cadena.lower(): Devuelve la cadena con los mismos caracteres que la cadena `cadena` pero en minúsculas.
- cadena.title(): Devuelve la cadena con los mismos caracteres que la cadena `cadena` con el primer carácter en mayúsculas y el resto en minúsculas.

[1]:
```
'Python'.upper()
```

[1]:
```
'PYTHON'
```

- cadena.split (delimitador): Devuelve la lista formada por las subcadenas que resultan de partir la cadena `cadena` usando como delimitador la cadena delimitador. Si no se especifica el delimitador utiliza por defecto el espacio en blanco.

[ ]:
```
'A,B,C'.split(',')
```

[ ]:
```
'I love Python'.split(' ')
```

3 Ipython

- Ipython https://ipython.org/ provee una poderosa shell interactiva para el lenguaje Python que además contiene:
  - un kernel para Jupyter https://jupyter.org/,
  - soporte para visualización interactiva de datos,

  - intérpretes embebidos para cargar en los proyectos, y
  - herramientas de alto desempeño para parallel computing https://ipyparallel.readthedocs.io/en/latest/.

- Para instalarlo en los sistemas operativos Debian/Ubuntu:

[ ]:
```
sudo apt install ipython3
```

- Para ejecutarlo:

[ ]:
```
ipython3
var='Hola Mundo'
print(var)
var?
```

- Escribiendo *var.* y tecleando el tabulador nos aparecerán los métodos de la clase *string* de nuestra variable.

[ ]:
```
var.upper()
```

- Para instalar Jupyter en Windows 10®:

[ ]:
```
python3 -m pip install --upgrade pip
python3 -m pip install jupyter
```

- Finalmente se ejecuta con el comando: *jupiter notebook*

3.1 IPython comandos útiles

[ ]:
```
?                -> Introducción a las características de IPython
%quickref        -> Referencia rápida
help             -> La ayuda de Python
object?          -> Detalles del 'object', con 'object??' se muestran detalles␣
 ↪extras
Esc + Enter       -> Salir de la ejecución del bloque de código
CTRL + o         -> Fuerza la creación de una nueva línea
Flechas arriba y abajo del teclado        -> Historia de los comandos realizados
Quit()               -> Salir
```

3.2 Magic functions en Ipython

- Hay dos tipos de "funciones mágicas" en Ipython: line magics y cell magics. https://ipython.readthedocs.io/en/stable/interactive/magics.html

- Line magics trabajan de manera muy similar a los comandos en el SO Linux o en DOS.

[ ]:
```
edit hola.py
%run hola.py # Ejecutar un script
!ls # En Linux
```

```
!dir # En Windows
```

[ ]:
```
fecha = !date
!echo "Hoy es {fecha}"
!echo $var
```

[ ]:
```
def hola():
    ...:     print("Hola Mundo")
hola()
```

- Para ver las variable y funciones definidas en IPython:

[ ]:
```
%whos
```

- Una práctica función es la que activa el *debugger* luego de un error en el código en modo "post mortem".
- Simplemente luego del error se ejecuta *%debug* para el *debug* interactivo.
- Cell magics a su vez, trabajan con múltiples líneas de código:

[ ]:
```
%%timeit elements = range(1000)
   ...: x = min(elements)
   ...: y = max(elements)
   ...:
   ...:
```

- Ejemplo usando la biblioteca timeit para medir el tiempo de ejecución de snippets de código.
- Extraido de https://switowski.com/blog/creating-magic-functions-part1.

3.3 Alias en IPython

- Para definir un alias para un comando utilizamos la sintaxis *%alias nombre_comando comando*

[ ]:
```
%alias? # Ayuda
%alias lll ls -la
lll
```

3.4 Jupiter Notebook

- Jupyter Notebook es una aplicación web *open source* que permite crear y compartir documentos.
- Los documentos pueden contener código en vivo, ecuaciones, visualizaciones y texto narrativo.
- Se utiliza entre otras cosas para:
  - Limpieza y transformación de datos,
  - Simulaciones numéricas,
  - Modelado estadístico,
  - Visualización de datos, y
  - Maching Learning.

- Para instalarlo:

[ ]:
```
pip install notebook
```

- Para ejecutar los cuadernos de *Jupyter*

[ ]:
```
jupyter notebook
```

- Se accede desde el navegador en la dirección web: http://localhost:8888
- El nuevo entorno de desarrollo que incluye a los cuadernos Jupyter se denomina JupyterLab.
- Instrucciones para instalarlo: https://jupyter.org/install