Práctica 1 - Jupyter como entorno y Python como lenguaje.

1. Entorno Jupyter

Jupyter Notebook es una aplicación web que permite crear y compartir documentos que contienen código fuente, ecuaciones, visualizaciones y texto explicativo. Entre sus usos está la limpieza y transformación de datos, la simulación numérica, el modelado estadístico, el aprendizaje automático y mucho más.

Jupyter está dividido en celdas. En cada una de ellas, se puede introducir texto, como ocurre en esta misma celda o código a ejecutar por el usuario, como se ha visto en los ejercicios del capítulo anterior.

La idea de usar Jupyter en las prácticas es la de seguir una progresión en el avance de los ejercicios combinando la teoría con la práctica.

Pulsando en la tecla >| Run se pasa a la siguiente, ejecutando el contenido de la celda en el caso de que hubiese código Python que ejecutar. En el caso de que únicamente haya texto (como en este ejemplo), simplemente se pasará a la siguiente.

Otra característica de Jupyter es que permite ejecutar de nuevo una celda ya ejecutada. En este curso, se ejecutarán celdas de código que repercutirán en el comportamiento del robot. Cualquier corrección necesaria podrá cambiarse en el momento y ese cambio será aplicado sobre el robot inmediatamente después de que le celda sea ejecutada de nuevo.

---

2. Lenguaje Python.

Python es uno de los lenguajes de programación dinámicos más populares que existen entre los que se encuentran Perl, Tcl, PHP y Ruby. Aunque es considerado a menudo como un lenguaje "*scripting*", es realmente un lenguaje de propósito general. En la actualidad, Python es usado para todo, desde simples "*scripts*", hasta grandes servidores web que proveen servicio ininterrumpido. Es utilizado para la programación de interfaces gráficas y bases de datos, programación web tanto en el cliente como en el servidor (véase Django o Flask) y "*testing*" de aplicaciones. Además tiene una amplia aceptación por científicos que hacen aplicaciones para las supercomputadores más rápidas del mundo y por los niños que recién están comenzando a programar.

- Lenguaje de alto nivel. Gramática sencilla, clara y muy legible.
- Tipado dinámico y fuerte.
- Código abierto.
- Fácil de aprender.
- Librería estandar muy amplia.
- Versátil. Es usado en multitud de disciplinas.

2.1.- Primer programa en Python ("Hola Mundo").¶

Abrimos el intérprete de Python tecleando python. Con el método print imprimimos por consola el elemento que vaya detrás:

```
print "Hola Mundo"
```

2.2.- Tipos, variables y operadores.

Tipos de datos en Python son:

- **Numéricos**: Enteros (int), con coma flotante (float), complejos.
- **Caracteres**: "Hola mundo".
- **Booleanos**: True, False.

Las variables son sencillas de declarar:

```
a = 1
b = 2
c = a + b
print c
3
```

```
a = 'hola'
b = 'mundo'
c = a + b
print c
'holamundo'
```

Algunos de los operadores de los que dispone las librerías integradas de Python son:

- Comentario: #
- Suma: +
- Resta: -
- Multiplicación: *
- División: /
- Módulo: %
- Exponente: **
- Igual que: ==
- Diferente que: !=
- Mayor y mayor o igual que: >, >=
- Menor y menor o igual que: <, <=

- Lógicos: and, or, not
- Asignación: =, +=, -=, ...
- Especiales: is, is not, in, ...

2.3.- Concepto de función.

Las aplicaciones informáticas que habitualmente se utilizan, incluso a nivel de informática personal, suelen contener cientos de miles de líneas de código fuente. A medida que los programas se van desarrollando y aumentan de tamaño, se convertirían rápidamente en sistemas poco manejables si no fuera por la modularización, que es el proceso consistente en dividir un programa muy grande en una serie de módlos mucho más pequeños y manejables.

A estos módulos se les suele llamar **funciones**. El lenguaje Python hace uso del concepto de función. La idea es siempre la misma: **dividir un programa grande en un conjunto de subprogramas** o funciones **más pequeñas** que son llamadas por el programa principal; estas a su vez llaman a otras funciones más específicas y así sucesivamente.

Las funciones en Python pueden recibir parámetros o argumentos y/o devolver valores. Es como una máquina expendedora: pasándole unos argumentos (el código del producto) retorna un valor (el producto seleccionado).

La sintaxis que se sigue es la siguiente:

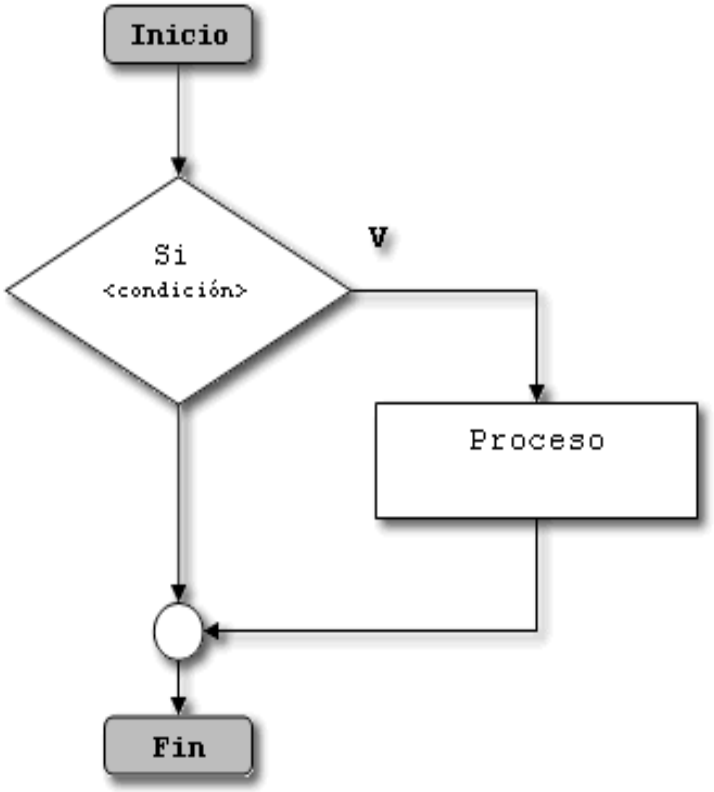
```
def nombre_funcion(parametros): # Los parámetros son opcionales.
    ...
    Instrucciones de la función
    ...
    return () # Opcional
```

Ejemplo de un programa principal y una función.

```
def suma(operando1, operando2):
    resultado = operando1 + operando2
    return resultado

...

# Programa principal

a = 1
b = 2
c = suma(a, b)
print c

# Fin del programa
```

2.4.- Condicionales.

Para escribir programas útiles, casi siempre necesitamos la capacidad de comprobar ciertas condiciones y cambiar el comportamiento del programa como corresponda. Las sentencias condicionales nos dan esta capacidad. La forma más sencilla es la sentencia if.



La estructura de control if ... permite que un programa ejecute unas instrucciones cuando se cumplan una condición. Si la condición **no** se cumple podemos construir un camino alternativo con la palabra reservada else.

```
|   instrucción                       |
|   instrucción                       |
|   instrucción                       |
|                                   |
|   if condición:                     |
|       intrucción si es True             |
|   else:                              |
|       instrucción si no es cumple (False)    |
|                                   |
|    instrucción                      |
|    instrucción                      |
\_/                                 \_/
```

Ejemplo de programa condicional.

Un ejemplo sencillo de programa condicional es este:

```
a = 2
b = 3
```

```
if a > 3: # ¿Es 2 > 3?
    print "La condición evaluada es True"
else:
    print "La condición evaluada es False"
```

Podemos añadir más condiciones en la misma estructura condicional con la palabra reservada elif. El ejemplo anterior con esta nueva funcionalidad quedaría así:

```
a = 2
b = 3

if a > 3: # ¿Es 2 > 3?
    print "La condición evaluada es True"
elif a < 3:
    print "La condición evaluada es False"
else:
    print "Son iguales"
```

2.6.- Listas (Arrays)

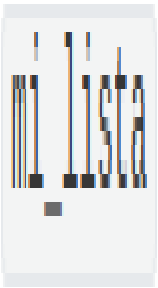
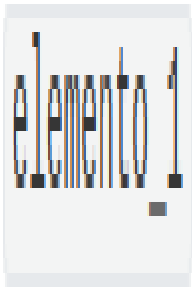
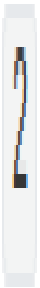
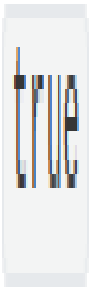
Las listas son un tipo de **colección ordenada** donde se pueden almacenar números enteros, con coma flotante, booleanos, cadenas de caracteres, etc.
Las listas nos sirven para almacenar varios elementos, por ejemplo, del mismo tipo, para luego poder ir recuperándolos buscando dentro. Esto evita tener que crear múltiples variables que tienen relación entre sí. Usando las listas, los datos quedan más estructurados.
Una lista vacía sería una tabla con una única fila. En Python se define como:

```
mi_lista = []
```

Podemos **añadir** elementos a la lista:

```
mi_lista = ["elemento_1", "hola", 2, True]
```

La lista se ordena a través de índices, donde el primer elemento tiene el índice 0. Podemos ver que es como una fila en una hoja de cálculo:

Se puede **agregar** elementos a la lista, utilizando el método append:

```
mi_lista.append("último")
```

Quedando:

```
["elemento_1", "hola", 2, True, "último"]
```

O el método insert para insertarlo en una posición concreta. Recuerda que el primer elemento de la lista tiene el índice 0.

```
mi_lista.insert(1, "elemento_insertado")
```

En este caso la lista queda:

```
['elemento_1', 'elemento_insertado', 'hola', 2, True, 'último']
```

De igual manera, se pueden **eliminar** elementos de una lista. Por ejemplo, con el método remove:

```
mi_lista.remove("hola")
```

Quedando como resultado:

```
['elemento_1', 'elemento_insertado', 2, True, 'último']
```

Otro método para eliminar elementos de una lista es, al igual que ocurría cuando se añadían campos, es utilizar el **índice** del elemento. Este método se llama pop() y hay que indicarle el índice. Vamos a eliminar de nuestra lista el elemento que hemos insertado en la celda anterior: elemento_insertado y que tiene el índice 1.

```
mi_lista.pop(1)
```

La lista tendrá ahora los siguientes elementos:

```
['elemento_1', 2, True, 'último']
```

2.7.- Bucles

En determinadas ocasiones será necesario repetir una tarea un número determinado o indeterminado de veces. Para no tener que repetir la misma orden una y otra vez a mano, los lenguajes de programación cuentan con una estructura de control para ello: los **bucles**.
Los bucles se encargan de repetir las tareas por nosotros. Podemos incluso poner condiciones (sentencias if) para que solo se ejecuten las órdenes si cumplen ciertos requisitos.
En esta sección se verán bucles for y bucles while.

2.7.1.- Bucles de tipo for

Los bucles for ejecutan las órdenes tantas veces (iteraciones) como nosotros le digamos, por ejemplo:

In [ ]:

```
for n in range(5):
    print("Hola, esta es la vuelta (iteración): " + str(n))
```

En este ejemplo se ha declarado una variable n que será la que lleve la cuenta de las vueltas o iteraciones. En cada una de las vueltas se imprime por pantalla una cadena de caracteres: "Hola, esta es la vuelta (iteración):". Además, a esta cadena se le añade la variable de iteraciones n que para nosotros será el número de iteración correspondiente. Esto da como resultado que se juntan la cadena de caracteres y la variable que contiene la vuelta del bucle.
Como puede verse, el número de vueltas se incrementa automáticamente y el bucle se detiene cuando llega al valor máximo, en este caso, 5.

2.7.2.- Bucles de tipo while

La principal diferencia con respecto al bucle for es que el número de vueltas no tiene por qué ser fijo ya que es el programador es el encargado de comprobar en cada iteración la condición de parada y de incrementar la variable de iteración. Veámoslo con un ejemplo.

Vamos a replicar el bucle anterior pero usando en este caso el bucle while. En primer lugar hay que **iniciar la variable de iteración** al valor inicial, normalmente 0. A continuación se entra en el bucle que tiene en la definición la **condición de parada**, por ejemplo, mientras la variable de iteraciones sea menor que 5, se quedará dentro. Por último queda **incrementar la variable** de iteración, sumandole uno en cada vuelta. Si se ejecuta el siguiente fragmento de código, puede verse como el resultado es el mismo.

In [ ]:

```
n = 0
while n < 5:
    print("Hola, esta es la vuelta (iteración): " + str(n))
    n += 1 # Esto incrementa en 1 la variable n
```

2.7.3.- Bucle infinitos ∞¶

Para el curso de Kibotics se utiliza el bucle infinito de tipo while. ¿Por qué while y no for?. Como ya se ha explicado, el bucle for da un número de vueltas predefinido mientras que con while podemos generar el bucle infinito poniendo que la condición de parada nunca se cumpla. En este curso, para la creación del bucle infinito se establece una condición del tipo: while True lo que fuerza al programa a quedarse dentro dado que no hay un número de iteraciones o una condición de parada. Vamos a verlo con un ejemplo.

Vamos a crear un bucle donde en cada iteración, el programa pida al usuario que escriba un número. El **bucle será infinito** y solo saldremos de él cuando escribamos el número 0. Para el resto de número, el bucle únicamente imprimirá por pantalla el valor que hayamos introducido. **Nota:** Introducir un caracter no numérico hará que el programa falle. **Solo acepta números**.

In [ ]:

```
while True:
    # El 0 hará detener el programa
    numero = input("Escribe un número: ")
    if int(numero) == 0:
        print("Introducido un 0. Fin del programa.")
        break
    print("El caracter introducido es: " + str(numero))
```

Con este tipo de bucles podemos tener programas que ejecuten únicamente una serie de órdenes en cada vuelta, como se verá en la programación de los robots.

2.8.- Librería Time

La biblioteca time contiene una serie de funciones relacionadas con la medición del tiempo. Podemos decirle a Python que ejecute una orden y espere un número de segundos a ejecutar la siguiente. Esto será necesario en el curso dado que los sensores leen muy rápido. Tanto que en ocasiones es demasiado rápido para el programa, por lo que hay que limitar esa velocidad. Se emplea para ello librerías de este tipo que controlan esa velocidad.

Veamos un ejemplo.

Este pequeño programa imprimirá por pantalla los números del 1 al 3 cada 2 segundos utilizando la librería time y una función que tiene esa librería, sleep. Como primer paso es importar la librería time. Luego se ejecuta el resto del programa.

In [ ]:

```
import time

print("Número 1")
time.sleep(2)
print("Número 2")
time.sleep(2)
print("Número 3")
time.sleep(2)
print("Fin del programa.")
```

2.9.- ¿Practicamos un poco?. Hagamos unos ejercicios.

Ejercicio 1: Definir una función max() que tome como argumento dos números y devuelva el mayor de ellos:

In [ ]:

```
def max (argumento1, argumento2):
    # Escribe tu código aquí
    if argumento1 > argumento2:
        mayor=argumento1
    elif argumento1 < argumento2:
        mayor=argumento2
    else:
        mayor=argumento1
    return mayor



# ----------------------------------------
# Programa principal

# Declarar dos variables
# variable1 = valor1
variable1= 5
# variable2 = valor2
variable2= 8

# Llama a la función e imprime el valor retornado.
print max(variable1, variable2)


# Fin del programa
```

Ejercicio 2: Escribir una función que tome un carácter y devuelva Truesi es una vocal, o False en caso contrario.

In [ ]:

```
def es_vocal (x):
    # return True o False
```

```
    if x =="a" or x =="e" or x =="i" or x =="o" or x =="u":
        return True
    else:
        return False

# Programa principal:
# letra = x
Letra=raw_input("escribe letra:")
if es_vocal (Letra):
    print "True"
else:
    print "False"



# Fin del programa
```

Ejercicio 3: Determinar si un numero esta entre 0 y 100

In [ ]:

```
def rango (x):
    # CÓDIGO DE LA FUNCIÓN
    if (x>0) and (x<100):
        return True
    else:
        return False

# ---------------------------------
x = input("introduce numero:  ")
if rango (x):
    print "True"
else:
    print "False"
    #Fin del programa
```

Ejercicio 4: Determinar si dos cadenas de texto son iguales.

In [ ]:

```
# Ejercicio 4
uno = raw_input ("Introduce la primera frase:    ")
dos = raw_input ("Introduce la primera frase:    ")
if uno == dos :
    print "son iguales"
else:
    print "No son iguales"
```

Ejercicio 5: Determinar si una cadena contiene una letra especifica.

In [ ]:

```
# Ejercicio 5
def frase (x) :
    if "t" in x :
        return "tiene t"
    else:
        return "no tiene t"
cadena = raw_input ("introduce una frase:  ")
print frase (cadena)
```

Ejercicio 6: Crear una lista que contenga nombres de frutas e imprimir por pantalla cada una de las frutas utilizando un bucle for.

In [ ]:

```
# Ejercicio 6
mi_lista = ["pera", "melocoton", "mandarina", "pomelo", "platano"]
for n in range (5) :
    print mi_lista [n]
```

Ejercicio 7: Crear una lista que contenga nombres de frutas e imprimir por pantalla cada una de las frutas utilizando un bucle while cada 3 segundos.

In [ ]:

```
# Ejercicio 7
import time
mi_lista = ["pera", "melocoton", "mandarina", "pomelo", "platano"]
n = 0
while n < 5:
    print mi_lista[n]
    n += 1
    time.sleep(3)
```

Ejercicio 8: Escribe una función que reciba una lista como parámetro y devuelva la longitud (número de elementos) de la lista.

In [ ]:

```
# Ejercicio 8
def longitud (x):
    dias = ["lunes", "martes", "miercoles", "jueves", "viernes", "sabado", "domingo"]
len (["lunes", "martes", "miercoles", "jueves", "viernes", "sabado", "domingo"])
```

Ejercicio 9: Escribe una función que tome una lista de números y devuelva la suma acumulada, es decir, una nueva lista donde el primer elemento es el mismo, el segundo elemento es la suma del primero con el segundo, el tercer elemento es la suma del resultado anterior con el siguiente elemento y así sucesivamente. Por ejemplo, la suma acumulada de [1, 2, 3] es [1, 3, 6].

In [ ]:

```
# Ejercicio 9
def lista ():
    numeros = [1,2,3,4]
    suma_numeros = []
    suma = 0
    for i in range (len(numeros)):
        suma += int (numeros[i])
        suma_numeros.append(suma)
    return suma_numeros
print (lista())
```

Ejercicio 10: Escribe una función ordenada que tome una lista de números enteros y/o flotantes y devuelva una nueva lista con los valores colocados en orden ascendente.

In [ ]:

```
# Ejercicio 10
mi_lista = [2,7,3,9,5]
mi_lista.sort()
print mi_lista
```

Referencias

- http://docs.python.org.ar/tutorial/3/index.html
- http://jupyter.org/documentation
- https://www.practicepython.org/