7. Python

En estos apuntes se presentan algunos aspectos destacados de Python y se dan algunos ejemplos de programas elaborados con el lenguaje.

7.1 Intérprete Python

Los programas escritos en Python normalmente son interpretados, esto es, se utiliza un intérprete del lenguaje para analizar y ejecutar cada instrucción de un programa. Las instrucciones pueden darse de manera interactiva o a través de un archivo que contendría un programa completo.

En la siguiente sección se dan algunos ejemplos de instrucciones dadas de manera interactiva las cuales también muestran algunos aspectos básicos del lenguaje (el texto que aparece con >>> es el que el usuario teclea; el que no tiene esos símbolos representa la respuesta del sistema).

Se utiliza el intérprete gratuito Spyder el cual puede ser bajado desde: https://www.spyder-ide.org/

7.2 Python interactivo

En esta sección se muestran algunas instrucciones básicas para introducir ciertas características del uso de los números y las cadenas en el lenguaje[1].

Números

Los comentarios en un programa se establecen con **#**.

Aquí se dan algunos ejemplos de la especificación de números y lo que debería contestar el intérprete (los puntos suspensivos (…) indican continuación de instrucción):

```
>>> #Esto es un comentario.
... 2+2
4
>>> 2+2   #Un comentario junto al código.
4
>>> (50-5*6)/4
5.0
>>>
```

[1] Varios de los ejemplos y descripción de las instrucciones están tomados de: Guido van Rossum y Fred L. Drake, Jr. (editor), "Guía de aprendizaje de Python", Release 2.0, BeOpen PythonLabs, 2000.

Los operadores aritméticos básicos son: +, -, *, /, // (división entera), % (módulo), ** (potencia). El módulo se define como: x % y = x – ((x // y) * y).

Ejemplos:

```
>>> 14 // 5
2
>>> 14 / 5
2.8
>>> 14 % 5
4
>>> 3.5 % 0.5
0.0
```

Se pueden asignar valores a variables y usarlas en instrucciones posteriores. Las variables **no necesitan** declararse previamente:

```
>>> ancho = 20
>>> alto = 5*9
>>> ancho * alto
900
>>> x = y = z = 0   #Poner a cero 'x', 'y' y 'z'
>>> x
0
>>>
```

Cadenas de caracteres

Se pueden especificar entre comillas o apóstrofos:

```
>>> "Éste es un ejemplo"
'Éste es un ejemplo'
>>> 'Éste es otro ejemplo'
'Éste es otro ejemplo'
```

Se pueden concatenar cadenas con el operador + y repetirlas con *:

```
>>> palabra = 'Ayuda' + 'Z'
>>> palabra
'AyudaZ'
>>> '<' + palabra*5 + '>'
'<AyudaZAyudaZAyudaZAyudaZAyudaZ>'
```

Se pueden manejar caracteres individuales de una cadena accediéndolos por medio de índices[2]:

```
+---+---+---+---+---+---+
| A | y | u | d | a | Z |
+---+---+---+---+---+---+
  0   1   2   3   4   5
```

En base a esto, los siguientes son ejemplos válidos:

```
>>> palabra[2]
'u'
```

[2] Los caracteres de una cadena, igual que en las listas, también pueden manejarse con índices negativos. Esta opción no se ve en estas notas.

```
>>> palabra[2:4]   #Se puede dar un rango de índices.
'ud'
>>> palabra[:2]    #Los primeros dos caracteres.
'Ay'
>>> palabra[2:]    #Todos menos los primeros dos caracteres.
'udaZ'
```

Esto es, cuando se da un índice solo, se toma el carácter indicado por el mismo. Cuando se da un rango de índices, se toman los caracteres que están dentro del rango, exceptuando el indicado por el segundo índice.

No se pueden cambiar caracteres dentro de una cadena:

```
>>> palabra[0] = 'x'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Sin embargo, se puede crear una nueva cadena usando las facilidades anteriores:

```
>>> 'x' + palabra[1:]
'xyudaZ'
```

Se pueden hacer concatenaciones usando los índices:

```
>>> palabra[:2] + palabra[2:]
'AyudaZ'
```

7.3 Instrucciones de control de flujo

El lenguaje brinda las instrucciones típicas para el control del flujo de ejecución de un programa. **La sangría en las instrucciones es significativa** y delimita al conjunto de instrucciones que están dentro de otra instrucción.

If

```
if condición:
      instrucciones
elif condición:
      instrucciones
elif condición:
      instrucciones
else:
      instrucciones
```

Ejemplo:

```
# -*- coding: utf-8 -*-

x = int(input("Introduce un número: "))
if x < 0:
      x = 0
      print('Negativo cambiado a cero')
elif x == 0:
      print('Cero')
elif x == 1:
      print('Uno')
else:
      print("Más")
```

Hay varias cuestiones que se pueden comentar acerca del código anterior:

1. La línea:
   ```
   # -*- coding: utf-8 -*-
   ```
   indica el tipo de codificación que se usará para el programa.
2. La función `input` permite leer un valor desde teclado. La función `int` guarda al valor como entero.
3. Los dos puntos (:) después del if y la sangría de sus siguientes dos líneas serían equivalentes a { } o begin-end en otros lenguajes.
4. La comparación por igualdad se hace con ==.
5. La palabra `elif` es equivalente a else if.

While

```
while condición:
      instrucciones
```

Ejemplo:

```
#! coding: latin-1

# Serie de Fibonacci.
a = 0
b = 1
print("Número: ",a)
print("Número: ",b)
while b < 20:
      temp = a + b
      a = b
      b = temp
      print("Número: ",b)
```

For

```
For variable in secuencia:
      instrucciones
```

La instrucción `for` cambia un poco con respecto a otros lenguajes ya que se aplica sólo a secuencias y se usa para recorrerlas en el orden en que aparecen en la secuencia. Una secuencia puede ser una **lista** o una cadena. Las listas, descritas más adelante, son secuencias de valores y, en principio, pueden verse como arreglos unidimensionales.

Ejemplo:

```
#! coding: latin-1

a = ['gato', 'ventana', 'casa']
for x in a:
      print(x, len(x))
```

La primera línea (de código) muestra cómo se crea la lista. La función `len(x)` obtiene la longitud de la cadena almacenada en `x`.

Nota: también se puede usar el operador *not in* dentro de un for.

7.4 Función range()

Sirve para generar una secuencia de números partiendo de un valor inicial hasta un valor final, sin incluirlo, con incrementos de uno en uno. Opcionalmente se pueden indicar incrementos/decrementos con valores diferentes de uno.

Ejemplos (nota: los resultados se verían como se muestra aquí, si el código se ejecutara en la versión 2.7):

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Ejemplos de for con range:

```
for x in range(1, 5):
      print(x)

1
2
3
4
```

```
for x in range(1, 11, 2):
      print(x)

1
3
5
7
9
```

7.5 Estructuras de datos

En esta sección se describen características importantes de las diferentes estructuras de datos que pueden usarse en Python.

7.5.1 Listas

Simplemente son listas de valores separados por comas y encerrados entre corchetes. Son similares a los arreglos de una dimensión, aunque los elementos de una lista no tienen que ser necesariamente del mismo tipo.

Como en las cadenas, los índices de una lista inician en cero; asimismo, se pueden emplear las diversas notaciones empleadas en las cadenas para especificar los índices de los elementos de una lista.

Ejemplos:

```
>>> a = ['nombre', 'apellido', 100, 1234]
>>> a
['nombre', 'apellido', 100, 1234]
>>> a[0]
'nombre'
>>> a[3]
1234
>>> a[1:3]
['apellido', 100]
>>> a[:2] + ['alias', 2*2]
['nombre', 'apellido', 'alias', 4]
```

A diferencia de las cadenas, cuyos valores no se pueden cambiar, **es posible remplazar los valores de los elementos de una lista**:

```
>>> a
['nombre', 'apellido', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['nombre', 'apellido', 123, 1234]
>>> len(a)
4
```

Se pueden cambiar, quitar y agregar elementos a una lista:

```
>>> # Cambiar elementos:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Quitar elementos:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Agregar elementos (pudiendo cambiar el tamaño de la lista):
... a = a + [67]
>>> a
[123, 1234, 67]
>>> a[1:1] = [321, 4321]
>>> a
[123, 321, 4321, 1234, 67]
```

Es posible anidar listas:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p
[1, [2, 3], 4]
>>> p[1]
[2, 3]
>>> p[1][0]
2
```

Con esta característica se puede crear el equivalente a arreglos de dos o más dimensiones:

```
>>> b= [[0,1], [2,3]]
>>> b
[[0, 1], [2, 3]]
>>> b[0]
[0, 1]
>>> b[1]
[2, 3]
>>> b[1][1]
3
```

El tipo "lista" tiene métodos asociados:

append(x) Añadir un elemento al final de una lista.

extend(L) Extender la lista concatenándole todos los elementos de la lista indicada.

insert(i, x) Inserta un elemento en un posición dada. El primer argumento es el índice del elemento antes del que se inserta, por lo que a.insert(0, x) inserta al principio de la lista y a.insert(len(a), x) equivale a a.append(x).

remove(x) Elimina el primer elemento de la lista cuyo valor es x. Provoca un error si no existe tal elemento.

index(x) Devuelve el índice del primer elemento de la lista cuyo valor sea x. Provoca un error si no existe tal elemento.

count(x) Devuelve el número de veces que aparece x en la lista.

sort() Ordena ascendentemente los elementos de la propia lista.

reverse() Invierte la propia lista.

Se puede usar `del` para eliminar elementos de una lista dando su índice o un rango de índices:

```
>>> c= [10,20,30,40,50]
>>> c
[10, 20, 30, 40, 50]
>>> del c[3]
>>> c
[10, 20, 30, 50]
>>> del c[1:3]
>>> c
[10, 50]
>>> del c
>>> c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

Al final marca error porque la lista ya no existe.

7.5.2 Tuplas

Una tupla es parecida a una lista, aunque tiene más el sentido de tupla en el modelo relacional: una serie de valores separados por comas. Ejemplos:

```
>>> t = 12345, 54321, '¡hola!'
>>> t[0]
12345
>>> t
(12345, 54321, '¡hola!')
>>> # Se pueden anidar tuplas:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, '¡hola!'), (1, 2, 3, 4, 5))
```

Las tuplas tienen las siguientes características:

- Al imprimirlas se muestran encerradas entre paréntesis.
- En la entrada los paréntesis son opcionales.
- Pueden anidarse.
- **No pueden cambiarse sus elementos**.
- **Pueden contener listas, las cuales sí pueden cambiarse**.

Tuplas de 0 o 1 elemento:

```
>>> vacio = ()
>>> singleton = 'hola', # <-- Obsérvese la coma final: necesaria.
>>> len(vacio)
0
>>> len(singleton)
1
>>> singleton
('hola',)
```

7.6 Funciones

Las funciones en Python son similares al concepto que se tiene de ellas en otros lenguajes de programación.

En Python se definen de la siguiente forma:

```
def  nombreFunción(parFormal_1, parFormal_2, …, parFormal_n):
    instrucción_1
    instrucción_2
         -----
    instrucción_m
    return resultado
```

donde:

parFormal_1, …, parFormal_n: pueden ser de cualquier tipo de datos. Opcionalmente se pueden escribir como: parFormal= *valor*, en cuyo caso el parámetro toma por default el *valor* indicado, si en el llamado a la función no se da valor para dicho parámetro.

instrucción_1, …, instrucción_m: pueden ser cualquier instrucción de Python.

return resultado: en el conjunto de instrucciones de la función debe haber al menos una instrucción *return* que regresa el resultado del proceso que hace la función.

Ejemplo: definición de la función que encuentra el máximo de dos valores:

```
def maxVal(x, y= 15):
    if x > y:
        return x
    else:
        return y
```

Llamados a la función:

```
>>> maxVal(20,40)
40
>>> maxVal(40,20)
40
>>> maxVal(10)
15
>>> maxVal(y=10, x=50)
50
```

```
>>> maxVal(y=10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: maxVal() missing 1 required positional argument: 'x'
>>> maxVal(x=10)
15
>>>
```

7.7 Diccionarios

Son conjunto no ordenados de pares clave:valor. Los pares se encierran entre {}, separando cada par con coma.

Ejemplo: `dic= {'a':10, 'b':20, 'c':30}`

Presentan las siguientes características:

- Las claves, en principio, pueden ser cadenas o números, aunque también pueden ser tipos inmutables como las tuplas.
- Las claves deben ser únicas dentro del mismo diccionario.
- El diccionario vacío se representa con {}.
- El acceso individual a elementos se hace como en una lista dando como índice la clave del elemento. Por ejemplo: `dic['a']` regresa el valor 10.
- Si se asigna un valor a un elemento cuya clave ya existe, entonces el nuevo valor sustituye al viejo. Por ejemplo: `dic['b']= 40`, pone 40 en el segundo elemento, perdiéndose el 20.
- Para agregar un nuevo elemento se escribe: `dic['d']= 50`, con lo que el diccionario quedaría como: `dic= {'a': 10, 'b': 40, 'c': 30, 'd': 50}`
- Aunque los valores anteriores se ven ordenados por clave, el conjunto no está ordenado.

Los métodos, principales, que se pueden usar con los diccionarios son:

- **len(dic)** regresa la cantidad de elementos de dic (4, según el último ejemplo).
- **list(dic.keys())** regresa una lista con las claves de dic.
- **list(dic.values())** regresa una lista con los valores de dic
- **k in dic** regresa True si k está en las claves de dic.
- **del(dic[k])** elimina el elemento (completo) con clave k.
- **for k in dic** itera sobre las claves de dic.
- **sorted(dic.keys())** regresa una lista con las claves ordenadas.

También:

- Con `dict()` se puede construir un diccionario:

```
dict([('sophie', 4139), ('guido', 4127), ('jack', 4098)])
```

- Se puede hacer:

```
{x: x**2 for x in (2, 4, 6)}
```

7.8 Entrada/salida

En esta sección se describen las instrucciones que se utilizan para leer/escribir datos, en su versión simple.

Lectura desde teclado

Se puede realizar usando el par de instrucciones indicadas en el siguiente código:

```
# Lee datos desde teclado, los guarda en una lista, los
# multiplica por 2 y los escribe en pantalla.

# Con estas dos instrucciones se lee desde teclado:
cad = input('Escribe números separados por comas: ')
arre = cad.split(',')

print('Los números de la lista multiplicados por 2 son:')
for num in arre:
      print(int(num) * 2)
```

Lectura y escritura con archivos

El siguiente programa muestra una lectura de datos y escritura de resultados empleando archivos de texto:

```
# Lee datos desde un archivo, los multiplica por 2 y
# los escribe en otro archivo.
print('Inicié lectura y escritura de archivos')

# Apertura de los archivos.
flec= open('Datos.txt','r')          #Con 'r' el arch. se abre para lectura.
fesc= open('Resultados.txt','w')     #Con 'w' se abre para escritura.

#Escritura en el archivo de salida.
fesc.write('Los números, y su multiplicación por 2, son: \n')

# Lectura desde archivo (cada línea se lee como cadena):
for línea in flec:
  listaNums = línea.split(',')      # Se crea una lista con los números leídos.

  for num in listaNums:             #Multiplica por 2 cada número.
    num2= float(num) * 2
    salida= str(num) + '\t' + str(num2) + '\n'
    fesc.write(salida)    #Escritura en el archivo de salida.

# Cierra archivos.
flec.close()
fesc.close()
print('Terminé lectura y escritura de archivos')
```

7.9 NumPy

Es el paquete fundamental para cómputo numérico en Python.

- Algunas de sus características principales son:
  - Brindar un arreglo multidimensional de objetos, *ndarray*, que proporciona operaciones aritméticas eficientes para la manipulación de arreglos.
  - Proporcionar funciones matemáticas para realizar operaciones rápidas sobre arreglos completos de datos sin tener que escribir ciclos.
  - Dar herramientas para leer/escribir arreglos de datos con archivos.
  - Brindar funcionalidades para realizar operaciones de álgebra lineal, manejo de números aleatorios y transformada de Fourier.

- Elementos básicos:
  - La clase principal de NumPy se llama **ndarray**, también conocida con el alias de **array**, que es un arreglo multidimensional de objetos.
  - Las dimensiones se llaman **ejes (axes)** y a la cantidad total de ejes se le conoce como **rango (rank)**.
  - Los principales atributos de un objeto ndarray son:
    - ndim – la cantidad total de dimensiones.
    - shape – las dimensiones del arreglo. Es un tupla de enteros indicando el tamaño del arreglo en cada dimensión. Para una matriz de *m* filas y *n* columnas, shape será (m, n).
    - size – la cantidad total de elementos del arreglo. Para el ejemplo anterior, size sería m*n.
    - dtype[.name] – el tipo de los elementos del arreglo.
  - Los principales métodos de un objeto ndarray son:
    - array(lista o tupla [, dtype= "tipo"]) – para crear un arreglo. Con *dtype* se puede especificar un tipo de datos concreto.
    - arange(n [, límite, inc/dec]) – genera un arreglo de una dimensión de hasta *n* elementos, con valores 0, 1, … Opcionalmente se puede dar un valor límite con determinado incremento/decremento, en cuyo caso se generarán menos elementos si el incremento/decremento es diferente de 1.
    - linspace(limInf, limSup, n) – parecido al anterior, generando n elementos con valores desde *limInf* hasta *limSup*.
    - reshape(m, n) – establece un arreglo de m * n elementos.
    - len(array) – la cantidad de elementos en la primera dimensión.
    - zeros(tupla [, dtype= "tipo"]) – crea un arreglo de ceros con la cantidad y tamaño de dimensiones indicada por *tupla*. Opcionalmente se puede dar un tipo de datos con *dtype*. El tipo default es float64.
      Ambos parámetros se aplican de manera similar para los dos métodos siguientes.
    - ones(tupla [, dtype= "tipo"]) – crea un arreglo de unos.
    - empty(tupla [, dtype= "tipo"]) – crea un arreglo con valores iniciales aleatorios.
    - dot (a, b) – hace la multiplicación de las matrices a y b. Alternativa: a.dot(b)
    - random.random(tupla) – arreglo de valores aleatorios en [0,1). Con *tupla* se indica la cantidad de dimensiones y la de valores en cada una.

- random.randint(limInf [, limSup, tupla]) – arreglo de valores enteros aleatorios en [limInf, limSup). Con *tupla* se indica la cantidad de dimensiones y la de valores en cada una.
- copy() – copia un arreglo creando otro nuevo.
- identity(n) – matriz identidad de n * n elementos.
- loadtxt("archivo" [, dtype= "tipo", delimiter= "carácter"]) – lee datos a un arreglo desde el *archivo* indicado. Cada fila en el archivo debe tener la misma cantidad de valores. El tipo de valores por default es *float*; el delimitador default es el *espacio*.
- savetxt("archivo", arreglo [, fmt="formato"]) – escribe en *archivo* el contenido de *arreglo*. Opcionalmente se puede dar un formato, ejemplo: "%6.2f", que indica que el arreglo contendría valores numéricos y se escribirían con 6 dígitos, dos de ellos decimales.

- Se pueden aplicar los siguientes operadores a los arreglos:
  - array operArit array – donde *operArit* es cualquiera de los operadores aritméticos básicos: + - * /. Las operaciones aplicadas son elemento a elemento.
  - array operArit valor – similar a la anterior, pero el segundo operando puede ser un *valor* numérico.
  - array operComp array – donde *operComp* es cualquiera de los operadores de comparación: < <= > >= == ¡=. El segundo operando también puede ser un *valor*. Igual que en los casos anteriores, las operaciones se aplican elemento a elemento.
  - array ** n – cada elemento del arreglo se eleva a la potencia *n*.

7.10 Pandas

Es el paquete fundamental para análisis de datos en Python.

- Algunas de sus características principales son:
  - Contiene estructuras de datos y herramientas de manipulación de datos diseñadas para hacer limpieza y análisis de datos en Python de manera rápida y fácil.
  - A diferencia de NumPy que está pensado para trabajar con arreglos de datos homogéneos, pandas fue diseñado para trabajar con datos heterogéneos (tabulares).

- Elementos básicos:
  - Sus dos estructuras de datos principales son: **Series** y **DataFrame**.
  - Una serie (Series) es similar a un arreglo unidimensional conteniendo una secuencia de valores (de tipos similares a los de NumPy) y un arreglo asociado de etiquetas, llamado su **índice**.
  - La serie más simple se forma a partir de un arreglo de datos.
  - Si no se especifica lo contrario, los índices van de 0 a *n*-1, siendo *n* la cantidad de valores de la serie.
  - Una serie creada a partir de un diccionario hace que las claves queden ordenadas ascendentemente. Si no se quiere este orden, hay que crear la serie especificando en el índice cómo van las claves.

pandas - Series

- Ejemplos

```
import pandas as pd
import numpy as np

#Definición de una serie. Sus índices son implícitos: [0,1,2,…]
#Los valores se pueden dar como lista o como tupla.
serie= pd.Series([4,7,-5,3])
serie    #Muestra su contenido: índices y valores.
serie.values   #Muestra sólo los valores.
serie.keys()   #Muestra los índices.
serie.index    #Ídem.

#Definición de una serie con índices explícitos (caracteres).
serie2= pd.Series([4,7,-5,3], ['d','b','a','c'])
serie2   #Los índices salen desordenados.
serie2.index
serie2['a']    #Contenido de un elemento.
serie2['d']=6  #Modificación de un elemento.
serie2[['c','a','d']]     #Listar el contenido de los tres elementos.
serie2[serie2>0]     #Muestra los valores > 0.
serie2*2       #Multiplicación escalar.
np.exp(serie2) #e^valores.
'b' in serie2  #Como en un diccionario.
```

```
#Serie creada a partir de un dicc.
sdata= {'gdl':4000, 'mty':4000, 'cdmex': 10000, 'qro': 1500}
serie3= pd.Series(sdata)

#Al crear una serie, se puede especificar cómo quedan los índices.
cds= ['león','mty','cdmex','gdl']
serie4= pd.Series(sdata,cds)
pd.isnull(serie4)    #Detecta nulos en la serie; da true/false.
pd.notnull(serie4)   #Detecta no nulos en la serie; da true/false.

#Ejemplo de algunos métodos de las series.
serie4.sort_index()  #Ordena la serie por los índices.
serie4.sort_values() #Ordena la serie por los valores.
serie4.max()         #Entrega el máximo valor.
```

pandas - DataFrame

- Un DataFrame representa una tabla rectangular de datos y contiene una colección ordenada de columnas, cada una de las cuales puede ser de un tipo de datos diferente (numérica, cadena, boolean, etc.). La estructura tiene índices tanto de filas como de columnas. Puede ser pensado como un conjunto de series, todas compartiendo los mismos índices.

- Ejemplos (usando el ej. 3.3)

```
import pandas as pd

#Crea un DataFrame
datos= {'Estado': ['EdoMex','CDMX','Veracruz','Jalisco'],
        'Pob': [17363,8811,8163,8110]}
inds=[15,9,30,14]
estados= pd.DataFrame(datos, inds)
estados              #Muestra su contenido: índices y valores.
estados.values       #Muestra los valores de la tabla.
estados.index        #Muestra los índices de las filas.
estados.columns      #Muestra el nombre de las columnas.
estados['Pob'] #Muestra los valores de una columna, junto con índices.
estados.loc[15]      #Fila de valores del índice 15.
estados.iloc[0]      #Fila de valores de la fila 0.
estados.iloc[0:3]    #Fila de valores de las filas 0 a 2.
estados['temp']=10   #Crea la columna 'temp'. Si se le asigna un único
   #valor, todas las filas tienen ese valor. Se puede asignar una lista.
del estados['temp']  #Borra la columna 'temp'.
estados.index.name='Clave'       #Dar nombre a los índices.
Los índices son inmutables y se comportan como un conjunto de tamaño
fijo.
```