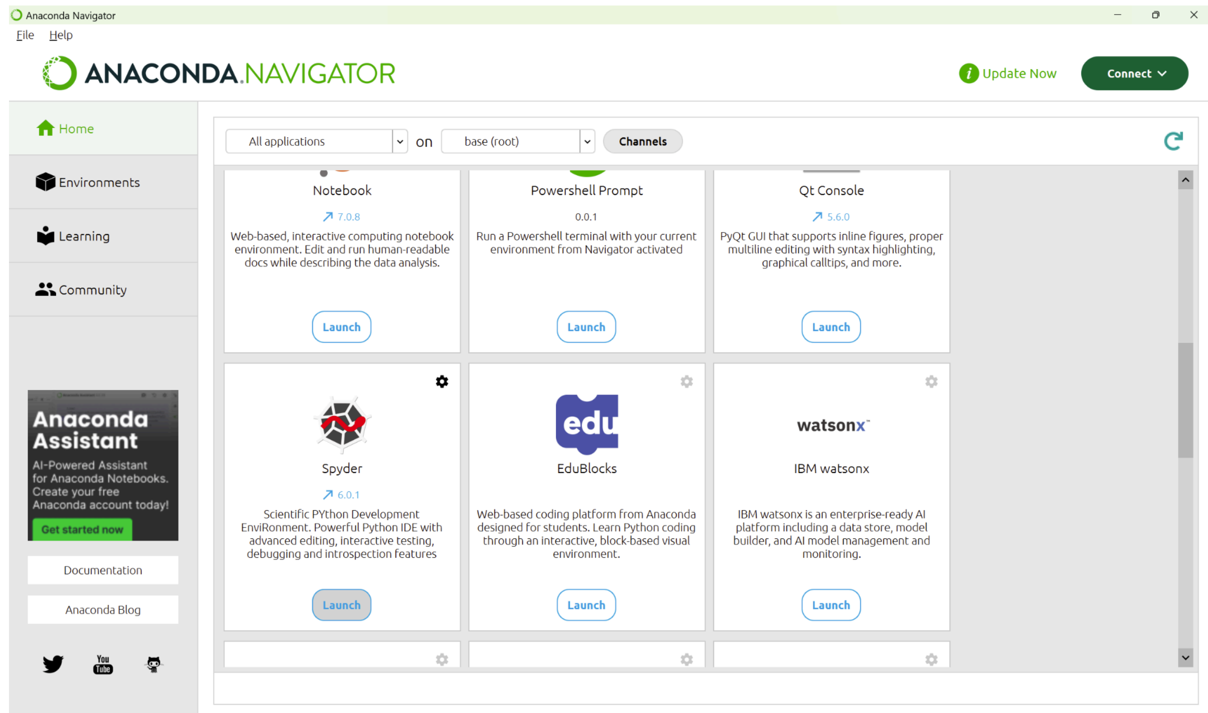
Tutorial Básico de Python

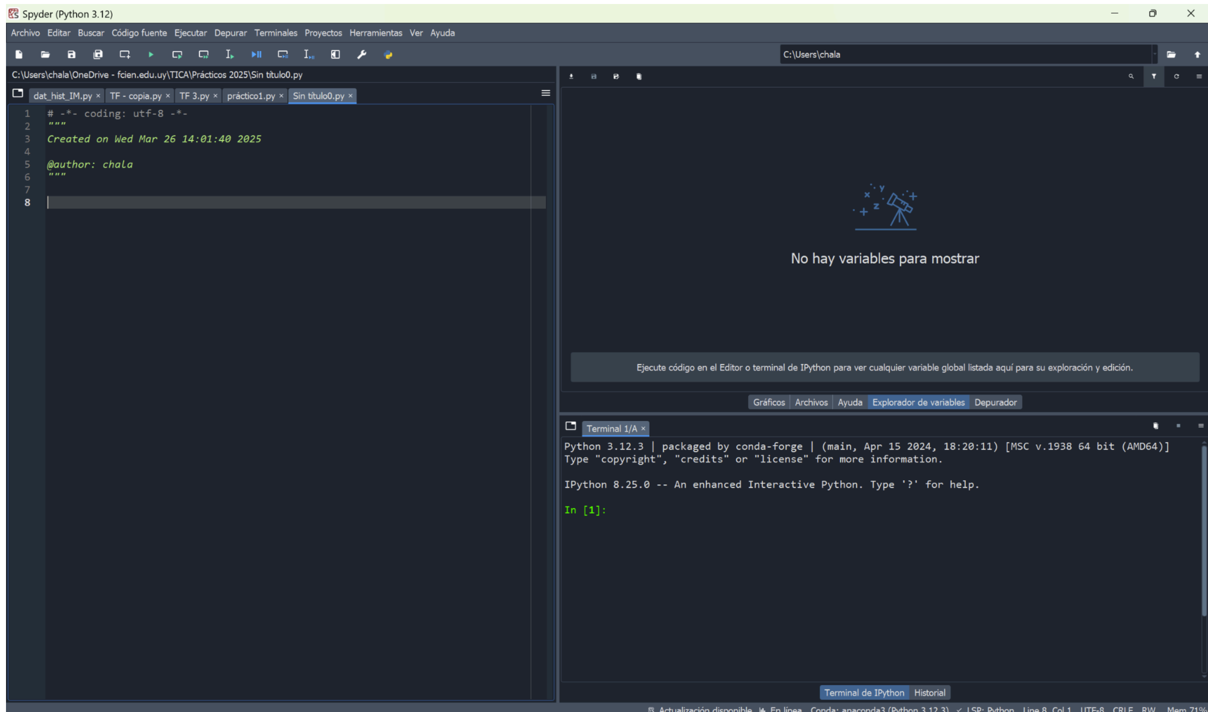
Python, Anaconda y spyder

Python es el lenguaje de programación que se utilizará a lo largo del curso, es amigable con principiantes y de código abierto, lo que lo hace una buena opción para comenzar. Anaconda es una plataforma de distribución de diversos programas que utilizan python, en particular nosotros utilizaremos “spyder”, un programa que, si bien no es de los más potentes es de uso muy simple.

Luego de descargar Anaconda (https://www.anaconda.com/download/success) y ejecutar el programa verán una ventana como la siguiente:



Lo único que se debe hacer para iniciar spyder es clickear en “launch” (es posible que spyder no esté instalado, en tal caso aparecerá un botón de “install” para instalarlo).



El entorno de spyer cuenta con 3 espacios: a la izquierda está el editor, donde se puede abrir, crear, editar y correr archivos, abajo a la derecha está la consola, que se puede utilizar tanto interactivamente como de forma indirecta a través del editor al correr los archivos, y arriba a la derecha está el panel de ayuda, que muestra distintos tipos de información útil en sus pestañas como los gráficos o las variables que creamos, o los archivos que se encuentran en nuestra computadora.

Nociones básicas.

Palabras Clave: Son aquellas que ya tienen asignada una función y no pueden ser usadas como nombre de una variable, etc.

Keywords in Python programming language				
False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

Operaciones básicas:
- Suma: a+b
- Resta: a-b
- Multiplicación: a*b
- División: a/b
- Potencia a**b
- Resto de una división: a%b
- Parte entera de una división: a//b

Paquetes: En python la gran mayoría de las funciones que se necesitan deben ser importadas primero por medio de “paquetes” que no son más que conjuntos de funciones típicamente relacionadas a un tema o área específica. Para importarlos se usa la palabra clave “import” seguida del nombre del paquete que se desea importar y si se desea se le puede asignar un nombre para utilizar mientras se trabaja agregando la palabra “as” seguida del nombre elegido. Por ejemplo, un paquete matemático muy útil es el paquete “numpy”.

(todas las líneas de código en este documento fueron corridas desde la consola).

```
In [1]: import numpy as np

In [2]: np.sin(45)
Out[2]: 0.8509035245341184
```

En este ejemplo importé el paquete “numpy” bajo el nombre “np” en la primera línea. En la segunda línea utilicé la función “sin” para hallar el seno de 45°. Siempre que se quiera utilizar una función de un paquete debe escribirse primero el nombre del paquete seguido de un punto seguido del nombre de la función.

Debido a que python es de código abierto pueden existir múltiples paquetes que cumplen funciones similares, los más usados suelen tener documentación extensa, tutoriales de buena calidad y gran cantidad de discusiones entorno a ellos en foros, esto hace que sea relativamente sencillo encontrar soluciones a los problemas con los que nos podamos encontrar.

Otro paquete muy útil es “matplotlib”, que se utiliza para graficar y crear figuras, en particular nos interesa el módulo “pyplot” que se encuentra dentro de ese paquete, para importarlo se escribe:

```
In [3]: import matplotlib.pyplot as plt
```

Variables

Crear una variable es tan sencillo como asignar un valor a un nombre que nosotros elijamos.

```
In [4]: a = 25
```

Así creamos una variable llamada “a” cuyo valor es 25. Este valor de a puede ser modificado ya sea sobreescribiéndolo o mediante modificadores:

```
In [4]: a = 25

In [5]: print(a)
25

In [6]: a+=1

In [7]: print(a)
26

In [8]: a=10

In [9]: print(a)
10
```

(con la función “print()” hago que python me muestre el valor de la variable entre paréntesis).

La lista de modificadores es

- Agregar una cantidad: a += b
- Restar una cantidad: a -= b
- Multiplicar por una cantidad: a *= b
- Dividir entre una cantidad: a /= b

También se pueden asignar más de una variable en una misma línea mediante el uso de comas:

```
In [10]: a,b = 1,2

In [11]: a,b
Out[11]: (1, 2)
```

Tipos de variables:

- int: Números enteros.
- float: Números reales, su precisión puede variar, en python se usa por defecto 64 bits que son 16 dígitos significativos.
- string: Líneas de texto.

El programa es capaz de identificar de qué tipo debe ser la variable al crearla y lo asigna automáticamente, aunque es posible cambiarlo manualmente.

Listas y arreglos (vectores)

- Listas: Conjunto de datos ordenados. Una lista con 10 datos tendrá índices del 0 al 9 y se define usando paréntesis rectos. Los paréntesis rectos también se usan luego del nombre de un arreglo o lista para seleccionar un elemento del mismo.

```
In [1]: lista = [4,6,1,7,9,7]

In [2]: lista[0]
Out[2]: 4

In [3]: lista[5]
Out[3]: 7
```

La función “len” nos permite saber el largo de una lista o arreglo:

```
In [4]: len(lista)
Out[4]: 6
```

Para modificar el contenido de un elemento de la lista lo reemplazamos de forma similar a una variable normal, los paréntesis rectos nos permiten seleccionar el elemento a modificar:

```
In [5]: lista[0]=10

In [6]: lista
Out[6]: [10, 6, 1, 7, 9, 7]
```

La función “append” permite añadir un valor nuevo a la lista:

```
In [7]: lista.append(100)

In [8]: lista
Out[8]: [10, 6, 1, 7, 9, 7, 100]
```

- Rangos: Con la función “range” podemos generar una secuencia de números enteros. Acepta de uno a tres argumentos: largo, inicio, paso, siendo sólo el primero obligatorio.

```
In [11]: rango1=list(range(10))

In [12]: rango1
Out[12]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [13]: rango2 = list(range(0,20,2))

In [14]: rango2
Out[14]: [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

In [15]: rango3 = list(range(10,100,10))

In [16]: rango3
Out[16]: [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

- Arreglos: Los arreglos funcionan como vectores, para construirlos utilizaremos el paquete “numpy”. La forma más fácil de construirlos es con la función “zeros” que permite definir el tamaño y las dimensiones del arreglo y lo construye con todas las entradas siendo cero.

```
In [19]: arreglo1 = np.zeros(5)

In [20]: arreglo1
Out[20]: array([0., 0., 0., 0., 0.])

In [21]: arreglo2 = np.zeros((2,2))

In [22]: arreglo2
Out[22]:
array([[0., 0.],
       [0., 0.]])
```

Después de eso debemos asignar los valores deseados a cada entrada a mano, en los índices siempre va primero la fila y luego la columna.

```
In [23]: arreglo2[1,0] = 1

In [24]: arreglo2
Out[24]:
array([[0., 0.],
       [1., 0.]])
```

Podemos combinar los arreglos con los rangos para acelerar el llenado de valores

```
In [25]: arreglo1[:] = range(1,6,1)

In [26]: arreglo1
Out[26]: array([1., 2., 3., 4., 5.])
```

El ":" indica que estoy referenciando a todos los elementos del arreglo. También sirve para referenciar un rango de índices.

```
In [27]: arreglo1[0:4] = 0

In [28]: arreglo1
Out[28]: array([0., 0., 0., 0., 5.])
```

Si multiplico un arreglo por otro del mismo tamaño con " * " la multiplicación se hará elemento a elemento y devolverá un arreglo del mismo tamaño

```
In [37]: arreglo1 = np.zeros(3)

In [38]: arreglo3 = np.zeros(3)

In [39]: arreglo1[:] = range(3)

In [40]: arreglo3[:] = range(3,6,1)

In [41]: arreglo1
Out[41]: array([0., 1., 2.])

In [42]: arreglo3
Out[42]: array([3., 4., 5.])

In [43]: arreglo1*arreglo3
Out[43]: array([ 0.,  4., 10.])
```

Los productos interno y vectorial se realizan con las funciones de "numpy" "dot" y "cross". Si el arreglo es de más de una dimensión "dot" realiza el producto matricial.

```
In [44]: np.dot(arreglo1, arreglo3)
Out[44]: 14.0

In [45]: np.cross(arreglo1, arreglo3)
Out[45]: array([-3.,  6., -3.])
```

Para operar con arreglos se les pueden aplicar todas las operaciones básicas como si el arreglo fuera un valor simple y la operación se aplicará a cada uno de los elementos del arreglo por separado.

```
In [42]: arreglo4 = np.zeros(5)

In [43]: arreglo4[:] = range(5)

In [44]: arreglo4
Out[44]: array([0., 1., 2., 3., 4.])

In [45]: arreglo4*2
Out[45]: array([0., 2., 4., 6., 8.])

In [46]: arreglo4**2
Out[46]: array([ 0.,  1.,  4.,  9., 16.])

In [47]: arreglo4-3
Out[47]: array([-3., -2., -1.,  0.,  1.])
```

Notar que en cada una de estas cuentas no estoy modificando el "arreglo4" simplemente estoy operando y mostrando el resultado, pero no lo estoy guardando.

Loops

Los loops permiten definir una variable llamada "iterador" que progresivamente cambia según una condición elegida por nosotros y nos permite realizar una función particular para cada una de esas iteraciones.

- for: "for" es el más sencillo, se le asigna al iterador un rango de valores y realiza una tarea para cada uno de ellos.

```
In [29]: texto = ['uno', 'tiempo', 'clima']
    ...: for i in texto:
    ...:     print(i, len(i))
    ...:
uno 3
tiempo 6
clima 5
```

```
In [30]: for i in range(5):
    ...:     print(i**2)
    ...:
0
1
4
9
16
```

- while: Este loop, en lugar de realizar un número predeterminado de veces, lo realiza hasta que se cumpla o deje de cumplir una determinada condición.

```
In [34]: a=0

In [35]: b=2

In [36]: while a<5:
    ...:     print(a)
    ...:     a += b//2
    ...:
0
1
2
3
4
```

La sintaxis de los loops es muy importante, en el primer renglón siempre se pone un ":" al final y todas las líneas de código que vayan dentro del loop deben estar indentadas (tabuladas) una vez (o una vez por nivel en caso de que se tenga un loop dentro de otro).

Condicionales y comparaciones

Estas funciones permiten realizar una acción siempre y cuando se cumpla una condición utilizando la palabra clave "if" seguida de la condición. Similarmente a los loops siempre se debe poner ":" al final de la primera línea y todas las líneas dentro del condicional deben estar indentadas. Para hacer las comparaciones los símbolos usados son:

- >,<: Mayor que, menor que.
- >=,<=: Mayor o igual que, menor o igual que
- ==: Igual que. Importante que cuando se quiere comparar se ponen dos símbolos de igual y no uno solo.
- !=: Distinto que

```
In [44]: a = 8

In [45]: if a == 8:
    ...:     print('a es igual a 8')
    ...:
a es igual a 8
```

También se pueden incluir qué hacer en caso de que no se cumpla la primera condición con la palabra clave "else".

```
In [49]: if a == 7:
    ...:     print('a es igual a 7')
    ...: else:
    ...:     print('a no es igual a 7')
    ...:
a no es igual a 7
```

Finalmente se pueden incluir condiciones alternativas en caso de que no se cumpla la condición anterior utilizando "elif".

```
In [50]: if a == 7:
    ...:     print('a es igual a 7')
    ...: elif a == 8:
    ...:     print('a es igual a 8')
    ...: else:
    ...:     print('a no es igual a 7 ni a 8')
    ...:
a es igual a 8

In [51]: a = 6

In [52]: if a == 7:
    ...:     print('a es igual a 7')
    ...: elif a == 8:
    ...:     print('a es igual a 8')
    ...: else:
    ...:     print('a no es igual a 7 ni a 8')
    ...:
a no es igual a 7 ni a 8
```

Leer datos de un archivo

Para leer datos de un archivo se necesita usar funciones de algún paquete. Para leer archivos .txt o .csv lo más sencillo es utilizar "loadtxt" de "numpy", Para poder utilizar

esta función es importante prestar atención a los detalles dado que si no especificamos ciertos argumentos la función debe asumirlos y puede dar errores.

- Ruta de acceso al archivo: Esta puede ser absoluta o relativa y debe estar siempre entre comillas.
- Delimitador: Por defecto es la coma, pero algunos archivos que se descarguen pueden estar utilizando otros delimitadores por lo que hay que revisarlo y especificarle a la función cuál debe utilizar. Algunos delimitadores utilizados además de la coma son el punto y coma (;), los dos puntos (:), las barras (/), los guiones (-) o la tabulación.
- Tipo de dato: Esto es importante dado que en ocasiones los archivos tendrán tanto datos de texto y numéricos, la función por defecto espera sólo numéricos, si se intenta cargar un archivo con texto sin especificarlo, la función devolverá un error. Dado que se pueden convertir datos numéricos a texto pero no al revés si se especifica el tipo de dato como “str” no dará error, pero no se pueden hacer cálculos matemáticos con variables de texto, por lo que habrá que convertirlos nuevamente más tarde.

```
In [53]: pres = np.loadtxt('inumet_presion_atmosferica_a_nive_del_mar.csv', delimiter=';', dtype=str)

In [54]: pres
Out[54]:
array([['fecha', 'estacion_id', 'pres_atm_mar'],
       ['', '', ''],
       ['2020-01-01 00:00', 'Rocha G3', '1005.8'],
       ...,
       ['', ' please try again later.', ''],
       ['', ' please try again later.', ''],
       ['', ' please try again later.', '']], dtype='<U24')
```

En este ejemplo importé un archivo de datos de presión atmosférica del INUMET, primero definí la ruta de acceso, en este caso relativa dado que el archivo de python está guardado en el mismo directorio que el archivo de datos, luego definí el delimitador como punto y coma y el tipo de dato como texto debido a que el archivo contiene fechas, nombre de estaciones, etc.

Módulo matplotlib.pyplot

Este módulo contiene la mayoría de las funciones básicas de graficación.

- .plot(x, y, …): Esta función permite hacer gráficos de líneas, con opciones para cambiar el estilo de la línea, el rango de los ejes, etc. El único argumento obligatorio es “x”.
- .hist(x, bins, …): Permite construir un histograma de la serie de datos “x”. “bins” define la cantidad de intervalos que tendrá el histograma pero no es obligatorio.
- .scatter(x, y, …): Funciona igual que “.plot” pero para gráficos de puntos.
- .bar(x, height, …): Permite crear gráficos de barras. “height” es el vector que contiene las alturas de las barras, “x” es la posición de las mismas.
- .contour/.contourf(x, y, Z, …): Permiten realizar gráficos de contorno. “x” e “y” son las coordenadas de los datos, Z debe ser una matriz de datos de 2 dimensiones con la misma cantidad de filas que el largo de “y” y la misma cantidad de columnas que el largo de “x”.

Otras funciones útiles

Algunas funcione útiles de “numpy” son:

- .mean(x, …): Calcula el promedio de los valores de la variable "x". Si "x" es de dos o más dimensiones el promedio se hace para todos los valores en todos los ejes a menos que se especifique lo contrario.
- .max/.min(x): Halla el valor máximo o mínimo de la variable "x".
- .where(condition, …): Halla el o los índices de un vector dado donde se cumple cierta condición.
- .arrange(a, …): Funciona igual que "range" pero acepta números reales y crea un array de numpy.
- .linspace(start, stop, num, …): Similar a ".arrange" pero en lugar de definir inicio, final y paso, se definen inicio, final y número de elementos que se desea y la función elige el paso apropiado.
- Funciones trigonométricas: "numpy" posee todas las funciones trigonométricas clásicas.

Otros paquetes útiles

Como fue mencionado python tiene cientos de paquetes distintos con gran variedad de funciones que facilitan la realización de ciertas tareas, algunos de estos paquetes útiles son:

- pandas: Facilita el trabajo con grandes bases de datos.
- netCDF4 y/o xarray: Permiten el trabajo con datos en formato ".nc" o "netCDF", formato típico de datos atmosféricos.
- scipy: Cuenta con módulos que facilitan el trabajo en distintas áreas de procesamiento matemático como la estadística, el álgebra lineal, etc.

Todos los paquetes mencionados cuentan con documentación y discusión extensiva online por lo que siempre será posible encontrar la solución a cualquier problema que se encuentre. Además una función muy útil es que si se escribe el nombre de la función o el módulo seguido de "?" spyder brindará una ventana con toda la documentación de dicha función o módulo.

```
In [79]: np.linspace?
```

```
Signature:
np.linspace(
    start,
    stop,
    num=50,
    endpoint=True,
    retstep=False,
    dtype=None,
    axis=0,
)
Call signature:  np.linspace(*args, **kwargs)
Type:            _ArrayFunctionDispatcher
String form:     <function linspace at 0x0000019AA96FA840>
File:            c:\users\chala\anaconda3\lib\site-packages\numpy\core\function_base.py
Docstring:
Return evenly spaced numbers over a specified interval.

Returns `num` evenly spaced samples, calculated over the
interval [`start`, `stop`].

The endpoint of the interval can optionally be excluded.

.. versionchanged:: 1.16.0
    Non-scalar `start` and `stop` are now supported.

.. versionchanged:: 1.20.0
    Values are rounded towards ``-inf`` instead of ``0`` when an
    integer ``dtype`` is specified. The old behavior can
    still be obtained with ``np.linspace(start, stop, num).astype(int)``
```

Presione **Q** para salir del paginador