Tema 0 - Clase 1 Introducción a `python 3`

Física Computacional

Para comenzar el trabajo con `python`, abrimos una terminal de comandos (para entornos linux/mxc) o una ventana de comandos si estamos trabajando bajo Windows.
Inmediatamente debemos de reconocer la versión de `python` que tenemos instalada y notamos que el prompt ya ha cambiado dejando ahora tres símbolos mayor que: >>> que nos dice que la terminal está lista para recibir comandos.

1. Operadores aritméticos.

1.1 Python como calculadora.

Una vez abierta la sesión en `python`, podemos aprovechar al máximo este lenguaje: contamos con una calculadora a la mano, sólo hay que ir escribiendo las operaciones en la línea de comandos.

```
In [32]: 3+4

Out[32]: 7

In [3]: 3/4

Out[3]: 0.75

In [4]: 3.0/4.0

Out[4]: 0.75

In [5]: 3//4

Out[5]: 0

In [6]: 4//3

Out[6]: 1

In [7]: 4/3

Out[7]: 1.3333333333333333

In [8]: 5.0 / 10 * 2 + 5

Out[8]: 6.0
```

¿por qué obtenemos este resultado??

```
In [9]: 5.0 / (10 * 2 + 5)

Out[9]: 0.2
```

Como podemos ver, el uso de paréntesis en las expresiones tiene una particular importancia sobre la manera en que se evalúan las expresiones.
Podemos elevar un número a una potencia en particular

```
In [11]: 2**3**2

Out[11]: 512
```

Vemos que elevar a una potencia, la manera en que se ejecuta la expresión se realiza en un sentido en particular: de derecha a izquierda

```
In [17]: (2**3)**2

Out[17]: 64
```

El uso de paréntesis nos indica que la expresión contenida dentro de ellos, es la que se evalúa primero, posteriormente se sigue la regla de precedencia de operadores.
El operador módulo $(\%)$ nos devuelve el residuo del cociente.

```
In [16]: 17%3

Out[16]: 2
```

2. Tabla de operadores.

Las siguientes son las operaciones matemáticas que se pueden realizar en `python 3.`

Operador	Operación	Ejemplo	Resultado
$**$	Potencia	$2 * *3$	8
$*$	Multiplicación	$7 * 3$	21
$/$	División	$10.5/2$	5.25
$//$	División entera	$10.5//2$	5.0
$+$	Suma	$3 + 4$	7
$-$	Resta	$6 - 8$	-2
$\%$	Módulo	$15\%6$	3

2.1 Precedencia de los operadores artiméticos.

1. Las expresiones contenidas dentro de pares de paréntesis son evaluadas primero. En el caso de expresiones con paréntesis anidados, los operadores en el par de paréntesis más interno son aplicados primero.
2. Las operaciones de exponentes son aplicadas después. Si una expresión contiene muchas operaciones de exponentes, los operadores son aplicados de derecha a izquierda.

3. La multiplicación, división y módulo son las siguientes en ser aplicadas. Si una expresión contiene muchas multiplicaciones, divisiones u operaciones de módulo, los operadores se aplican de izquierda a derecha.
4. Suma y resta son las operaciones que se aplican por último. Si una expresión contiene muchas operaciones de suma y resta, los operadores son aplicados de izquierda a derecha. La suma y resta tienen el mismo nivel de precedencia.

3. Operadores relacionales.

Cuando se comparan dos (o más expresiones) mediante un operador, el tipo de dato que devuelve es lógico: **True** o **False**, que también tienen una representación de tipo numérico:

- **True** = 1
- **False** = 0

```
In [1]: 1 + 2 > 7- 3

Out[1]: False

In [33]: 1 < 2 < 3

Out[33]: True

In [ ]: 1 > 2 == 2 < 3

In [34]: 1 > (2 ==  2) <3

Out[34]: False

In [3]: 3 > 4 < 5

Out[3]: False

In [5]: 1.0 / 3 < 0.3333

Out[5]: False

In [6]: 5.0 / 3 >= 11 /7.0

Out[6]: True

In [8]: 2**(2. /3) < 3**(3./4)

Out[8]: True
```

3.1 Tabla de operadores relacionales.

Operador	Operación	Ejemplo	Resultado
$==$	Igual a	$4 == 5$	`False`
$!=$	Diferente de	$2! = 3$	`True`
$<$	Menor que	$10 < 4$	`False`
$>$	Mayor que	$5 > -4$	`True`
$<=$	Menor o igual que	$7 <= 7$	`True`
$>=$	Mayor o igual que	$3.5 >= 10$	`False`

3.2 Operadores boleanos.

En el caso del operador boleano `and` y el operador `or` evalúan una expresión compuesta por dos (o más términos), en ambos operadores se espera que cada término tenga el valor de `True`, en caso de que esto ocurra, el valor que devuelve la evaluación, es `True`, como se verá en la tabla de verdad, se necesita una condición particular para que el valor que devuelva la comparación, sea `False`.

Operador	Operación	Ejemplo	Resultado
`and`	conjunción	`False and True`	`False`
`or`	disyunción	`False or True`	`True`
`not`	negación	`not True`	`False`

3.3 Tabla de verdad de los operadores boleanos.

En la siguiente tabla se comparan dos expresiones: `A` y `B`, en cada renglón se indica el posible valor booleano que puede tomar cada una de las expresiones, así como el resultado de utilizar el operador `and` y el operador `or`, en la última columna, se muestra el resultado de aplicar el operador `not` a la expresión `A`.

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

4. Tipos de variables.

Las variables en `python` sólo son ubicaciones de memoria reservadas para almacenar valores. Esto significa que cuando se crea una variable, se reserva un poco de espacio disponible en la memoria.
Basándose en el tipo de datos de una variable, el intérprete asigna memoria y decide qué se puede almacenar en la memoria reservada. Por lo tanto, al asignar diferentes tipos de datos a las variables, se pueden almacenar **enteros**, **decimales** o **caracteres (cadenas)** en estas variables.

4.1 Asignando valores a las variables.

Las variables de `python` no necesitan una declaración explícita para reservar espacio de memoria. La declaración ocurre automáticamente cuando se asigna un valor a una variable.

El signo igual (=) se utiliza para asignar valores a las variables.

El término a la izquierda del operador = es el *nombre de la variable* y el término a la derecha del operador = es el *valor almacenado* en la variable.
Nota: en las siguientes instrucciones, ocurren varias cosas: hasta el momento, escribimos una instrucción en la terminal y ésta se ejecuta al momento de teclear Enter, pero es posible escribir más de una instrucción por lo debemos de "avisarle" a `python` mediante el uso de un carácter especial para que entienda que aún no se termina de teclar: se usan los caracteres (`;\`), lo que nos dejará identada la siguiente línea y mostrará tres puntos, que es la señal de `python` para continuar tecleando alguna instrucción:

```
>>> contador = 100 ;\
...  distancia = 1000.0 ;\
... nombre = "Chucho"
```

En la última instrucción se teclea Enter para avisarle a `python` que ya concluimos de ingresar las instrucciones, y podemos pedirle ahora que nos muestre el contenido de las variables:

```
>>> print (contador) ;\
... print (distancia) ;\
... print (nombre)
100
1000.0
Chucho
```

4.2 Asignación múltiple de valores.

En `python` podemos asignar un valor único a varias variables simultáneamente.

```
In [8]: A = b = c = 1
        print (A)
        print (b)
        print (c)

1
1
1
```

En el ejemplo, se crea un objeto entero con el valor 1, y las tres variables se asignan a la misma ubicación de memoria.
También se puede asignar varios objetos a varias variables.

```
In [9]: A, b, c = 1, 2, "Alicia"
        print (A)
        print (b)
        print (c)

1
2
Alicia
```

Aquí, dos objetos enteros con valores 1 y 2 se asignan a las variables A y b respectivamente, y un objeto de cadena con el valor **Alicia** se asigna a la variable c.

4.3 Tipos de Datos Estándar.

Los datos almacenados en la memoria pueden ser de varios tipos. Por ejemplo, la edad de una persona se almacena como un valor numérico y su dirección se almacena como caracteres alfanuméricos.
En `python` se cuenta con varios tipos de datos estándar que se utilizan para definir las operaciones posibles entre ellos y el método de almacenamiento para cada uno de ellos.
Los tipos de datos son cinco:

1. Números.
2. Cadena.
3. Lista.
4. Tupla.
5. Diccionario.

4.3.1 Números

Los tipos de datos numéricos almacenan valores numéricos. Los objetos numéricos se crean cuando se les asigna un valor.

```
In [14]: Var1 = Var2 = 10
         print (Var1)
         print (Var2)

10
10
```

También se puede eliminar la referencia a un objeto numérico utilizando la sentencia `del`.

La sintaxis de la sentencia `del` es:

$del\ var1[, var2[, var3[...., varN]]]$

Se puede eliminar un solo objeto o varios objetos utilizando la sentencia `del`
Por ejemplo:

$del\ var$

$del\ variable1, variable2$

En `python` se soportan tres tipos numéricos diferentes:

1. Int (enteros con signo)
2. Flotante (valores reales de punto flotante)
3. Complejos (números complejos)

Un número complejo consiste en un par ordenado de números reales de coma flotante denotados por

$$x + yj$$

donde x e y son números reales, yj es la unidad imaginaria. Todos los enteros en `python 3` se representan como enteros largos. Por lo tanto, no hay ningún tipo de número por separado.

Ejemplos de tipo de números

int	float	complex
10	0.0	$3.14j$
100	15.20	$45.j$
080	$32.3 + e18$	$0.876j$
-0490	$-90.$	$-.645 + 0j$
$-0x260$	$70.2 - E12$	$3e + 26j$
$0x69$	$-32.54e100$	$4.53e - 7j$

4.3.2 Cadenas

Las cadenas en `python` se identifican como un conjunto contiguo de caracteres representados en las comillas. Con `python` se permite cualquier par de comillas simples o dobles: "hola", 'hola'

Los subconjuntos de cadenas pueden ser tomados usando el operador de corte ([ ] y [:]) con índices comenzando en 0 al inicio de la cadena hasta llegar a -1 al final de la misma.

El signo más $(+)$ es el operador de concatenación de cadenas y el asterisco $(*)$ es el operador de repetición.

```
In [15]: cadena = 'Hola Mundo!'

         print (cadena)           # Presenta la cadena completa
         print (cadena[0])        # Presenta el primer caracter de la cadena
         print (cadena[2:5])      # Presenta los caracteres de la 3a a la 5a posicio
         print (cadena[2:])       # Presenta la cadena que inicia a partir del 3er c
         print (cadena * 2)       # Presenta dos veces la cadena
         print (cadena + "PUMAS") # Presenta la cadena y concatena la segunda caden

Hola Mundo!
H
la
la Mundo!
Hola Mundo!Hola Mundo!
Hola Mundo!PUMAS
```

4.3.3 Listas

Las listas es el tipo de dato más versátil de los tipos de datos compuestos de `python`.

Una lista contiene elementos separados por comas y entre corchetes ([ ]). En cierta medida, las listas son similares a los arreglos (arrays) en el lenguaje C. Una de las diferencias entre ellos es que todos los elementos pertenecientes a una lista pueden ser de tipo de datos diferente.

Los valores almacenados en una lista se pueden acceder utilizando el operador de división ([ ] y [:]) con índices que empiezan en 0 al principio de la lista y opera hasta el final con -1.

El signo más (+) es el operador de concatenación de lista y el asterisco (∗) es el operador de repetición.

```
In [17]: milista = [ 'abcd', 786 , 2.23, 'salmon', 70.2 ]
         listabreve = [123, 'pizza']

         print (milista)              # Presenta la lista completa
         print (milista[0])          # Presenta el primer elemento de la lista
         print (milista[1:3])        # Presenta los elementos a partir de la 2a pos
         print (milista[2:])          # Presenta los elementos a partir del 3er ele
         print (listabreve * 2)       # Presenta dos veces la lista
         print (milista + listabreve) # Presenta la lista concatenada con la segund

['abcd', 786, 2.23, 'salmon', 70.2]
abcd
[786, 2.23]
[2.23, 'salmon', 70.2]
[123, 'pizza', 123, 'pizza']
['abcd', 786, 2.23, 'salmon', 70.2, 123, 'pizza']
```

4.3.4 Tuplas

Una tupla es otro tipo de datos de secuencia que es similar a la lista. Una tupla consiste en un número de valores separados por comas. Sin embargo, a diferencia de las listas, las tuplas se incluyen entre paréntesis.

La principal diferencia entre las listas y las tuplas son:

1. Las listas están entre corchetes [ ] y sus elementos y tamaño pueden cambiarse.
2. Las tuplas están entre paréntesis ( ) y no se pueden actualizar.

Las tuplas pueden ser consideradas como listas de sólo lectura.

```
In [18]: mitupla = ( 'abcd', 786 , 2.23, 'arena', 70.2  )
         tuplabreve = (123, 'playa')

         print (mitupla)                # Presenta la tupla completa
         print (mitupla[0])             # Presenta el primer elemento de la tupla
         print (mitupla[1:3])           # Presenta los elementos a partir del 2o el
         print (mitupla[2:])            # Presenta los elementos a partir del 3er e
         print (tuplabreve * 2)         # Presenta dos veces la tupla
         print (mitupla + tuplabreve)   # Preseta la tupla concatenada con la otra

('abcd', 786, 2.23, 'arena', 70.2)
abcd
(786, 2.23)
(2.23, 'arena', 70.2)
(123, 'playa', 123, 'playa')
('abcd', 786, 2.23, 'arena', 70.2, 123, 'playa')
```

El siguiente código es inválido con la tupla, porque intentamos actualizar una tupla, que la acción no está permitida. El caso es similar con las listas.

```
In [19]: mitupla = ( 'abcd', 786 , 2.23, 'edificio', 70.2  )
         milista = [ 'abcd', 786 , 2.23, 'energia', 70.2  ]
         mitupla[2] = 1000    # Sintaxis invalida para la tupla
         milista[2] = 1000     # Sintaxis invalida para la lista


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-19-a99f473d7b8f> in <module>()
          1 mitupla = ( 'abcd', 786 , 2.23, 'edificio', 70.2  )
          2 milista = [ 'abcd', 786 , 2.23, 'energia', 70.2  ]
    ----> 3 mitupla[2] = 1000    # Sintaxis invalida para la tupla
          4 milista[2] = 1000     # Sintaxis invalida para la lista


        TypeError: 'tuple' object does not support item assignment
```

Pero en la lista podemos agregar nuevos elementos que se colocan al final de la misma:

```
In [21]: print(milista)
         milista.append('hola')
         print(milista)

['abcd', 786, 2.23, 'energia', 70.2]
['abcd', 786, 2.23, 'energia', 70.2, 'hola']
```

4.3.5 Diccionarios

Los diccionarios de `python` son de tipo tabla-hash.

Funcionan como arrays asociativos y consisten en pares de emphclave-valor.

Una clave de diccionario puede ser casi cualquier tipo de `python`, pero suelen ser números o cadenas. Los valores, por otra parte, pueden ser cualquier objeto arbitrario de `python`.

Los diccionarios están encerrados por llaves { } y los valores se pueden asignar y acceder mediante llaves cuadradas [ ].

```
In [30]: fisicos = dict()

         fisicos ={
             1 : "Eistein",
             2 : "Bohr",
             3 : "Pauli",
```

```
            4 : "Schrodinger",
            5 : "Hawking"
        }

        print(fisicos)
        print (fisicos.keys())
        print (fisicos.values())
        fisicos["6"] = "Planck"    #agrega un nuevo elemento al diccionario, tanto
        print(fisicos)
```

```
{1: 'Eistein', 2: 'Bohr', 3: 'Pauli', 4: 'Schrodinger', 5: 'Hawking'}
dict_keys([1, 2, 3, 4, 5])
dict_values(['Eistein', 'Bohr', 'Pauli', 'Schrodinger', 'Hawking'])
{1: 'Eistein', 2: 'Bohr', 3: 'Pauli', 4: 'Schrodinger', 5: 'Hawking', '6': 'Planck'
```

4.4 Regla para los identificadores.

Los identificadores son nombres que hacen referencia a los objetos que componen un programa: **constantes**, **variables**, **funciones**, etc.

Reglas para construir identificadores:

- El primer carácter debe ser una letra o el carácter de subrayado (guión bajo)
- El primer carácter puede ir seguido de un número variable de dígitos numéricos, letras o carácteres de subrayado.
- No pueden utilizarse espacios en blanco, ni símbolos de puntuación.
- En `python` se distingue de las mayúsculas y minúsculas.
- No pueden utilizarse palabras reservadas del lenguaje.

Lista de palabras reservadas en `python`:

```
  and       assert      break       class       continue    def
del         elif        else        except      exec        finally
for         from        global      if          import      in
is          lambda      not         or          pass        print
raise       return      try         while       yield       del
```