1 Comentarios - Markdown

En python como cualquier otro lenguaje, existen comandos que permiten poner comentarios que no afectan la corrida del programa pero que son muy útiles en la documentación del código realizado por el programador.

Veamos su uso:

A partir de un "#" todo lo que sigue es un comentario que se acaba cuando termina el renglón. Puede estar en la mitad de un comando.

En cambio tres " producen comentarios que pueden ocupar varios renglones. Tres comillas (3") lo inician al comentario y otras tres comillas lo terminan.

Ejemplo

""" Este en cambio es un comentario que ocupa muchas líneas """ <— A este tipo de comentarios se los denomina docstrings

Por supuesto que estos tipos de comentarios son para celdas de código. Pero si utilizo un **Notebook** para escribir y correr un programa en Python tendré más opciones. En un notebook tenemos dos tipos de celdas. Unas para el escribir código y otra para comentarios más sofisticados. Este tipo de celda es para utilizar el lenguaje de texto Markdown. Que es una versión simplificada del **Latex** que permite incluso poner fórmulas utilizando este último lenguaje. Resumiendo tenemos dos tipos de celdas: las de código, donde escribo partes de mi programa y las de comentarios escritas en legunaje Markdown. Esto sólo es válido si desarrollan y corren programas en lenguaje Python en un Notebook.

2 Variables en Python y manejo de datos

2.1 Variables básicas

Las variables de las sentencias python tienen una diversidad muy grande, están las básicas del python original, pero que se complementan con otras que son agregadas por distintas librerías (numpy, pandas, astropy, etc..). Muchas de estas son de uso muy común en el análisis de datos. Dado esta diversidad, para evitar confusiones existen órdenes para preguntar a una variable de qué tipo es (por ejemplo: type()). Veremos cada una de ellas, empezando por las básicas que son muy parecidas a las de los otros lenguajes y son las que vimos en el fortran determinadas por el Hardware de las computadoras. Pero hay siempre que tener presente que Python es un lenguaje orientado a objetos, así que las variables son objetos. Y que los objetos pueden ser creados por el usuario, es decir, uno puede crear variables con las propiedades que se necesiten creando una clase que incluya la definición adecuada.

Tipos de datos básicos:

- Enteros (Integers)
- Flotantes (Floats) incluye números complejos
- Textos (Strings)
- Lógicos (Boolean)
- Números complejos

2.1.1 Números

Veamos como asigno un número en este caso un "1" a la variable python "uno". Para probar que esto sucedió imprimiremos el número a continuación. Por lo cual la celda tiene dos comandos, la asignación del valor y la impresión de la variable.

[1]:
```
uno = 1
print("En la variable uno se asignó: ",uno)
```

```
En la variable uno se asignó:  1
```

Noten que no definí el tipo de variable, esto se hace automáticamente y se realiza en la asignación que se haga sobre esta variable en particular

En nuestro caso:

uno = 1

Así sería entero (ojo no es 1. -> no tiene el ".")

Resumiendo: una variable se crea cuando se le asigna un número, un texto, un resultado booleano. Esa característica de lo que se le asigna determina el tipo de variable que es de ahora en adelante. En este caso es un entero, entonces "uno" es una variable que guarda en si números enteros.

El comando print() imprimió el resultado en la zona próxima, abajo de la celda. Resumiendo, tenemos las celdas con las órdenes y luego fuera de la celda los resultados de la corrida de esos comandos.

¿Qué hago si en mitad de un programa quiero preguntar cuál es el tipo de una variable en particular?

Para eso tengo la orden type

En este caso es un entero. Hay que prestar atención a la orden print(), en la cual el texto puede ponerse con ¨ (o también '), para que se imprima un texto como tal cual es. Las distintas variables se separan con una coma (",") al igual que en Fortran.

[2]:
```
print('Esa variable es del tipo', type(uno))
```

```
Esa variable es del tipo <class 'int'>
```

[3]:
```
# es decir, puedo hacer lo siguente:

uno_f =1.
print(uno,uno_f)

# El 1 flotante es 1.0 (con el . y el 0, mientras que en el número
# entero esto no sucederá)

print('Esa variable es del tipo', type(uno_f))
```

```
1 1.0
Esa variable es del tipo <class 'float'>
```

Existen nombres que son prohibidos, no se permite su uso como variables o funciones, ya que identifican órdenes ya existentes del lenguaje Python. Este es el listado de esos nombres:

and, as, 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, with, yield

True, False, None <– Estas últimas tres son las únicas en mayúsculas

En los nombres de las variables **no pueden** estar estos símbolos: **!, @, #, $, %**

El nombre de una variable no puede empezar con un número, pero pueden tenerlo dentro del nombre. Se pueden usar variables con nombres largos, y a veces es cómodo separar las palabras con el símbolo (_) que puede ser usado como parte del nombre de la variable

Que_feo_numero = 171717171.71717

2.1.2 Sentencias multilínea

Hay que prestar atención a la sangría ("identation" en inglés) y a los símbolos: ( ), [ ] y { }. La sangría en Python significa que la sentencia está relacionada o englobada en la orden anterior. A diferencia de los demás lenguajes de programación su uso indica una acción, no es "inerte". Tiene significado funcional e indica una interacción con órdenes anteriores. Dicho de otra manera, es parte activa de una orden anterior junto con las otras sentencias que comparten el mismo tamaño de sangría. Puede haber dentro de un grupo de órdenes con sangría determinada, otras con un espaciado más largo. Estas últimas forman otro de grupo de órdenes relacionadas en un entorno más interno.

[2]:
```
a = 1 + 2 + 3 \
    + 4 + 5 + 6 \
    + 7 + 8 + 9

b = (1 + 2 + 3
    + 4 + 5 + 6
    + 7 + 8 + 9)

print("El valor de a=",a)
```

```
El valor de a= 45
```

También se pueden iniciar muchas variables en un sólo renglón

[5]:
```
a = 1; b = 2; c = 3

x = y = z = "Lo mismo en las tres variables."

print(x,y)
```

```
Lo mismo en las tres variables. Lo mismo en las tres variables.
```

2.1.3 Variables de Caracteres (Strings)

Los strings son cadenas de caracteres de texto, y sirven para manejar justamente textos. Se les pueden aplicar muchos comandos y funciones para realizar distintas modificaciones. Es posible cortar o pegar textos, en la sección de operaciones las veremos con detalle.

[6]:
```
curso = 'Curso2022'

print(curso)
```

```
Curso2022
```

Puedo ir pidiendo las letras que forman la palabra de a una, pero tenemos que tener cuidado sobre cómo se numeran las componentes de una estructura en Python Veamos si pido el elemento “0” y NO el “1”

Fijense (muy importante que uso los corchetes y no paréntesis)

[7]:
```
curso[0]  # El primer elemento es el "0" !!!
```

[7]:
```
'C'
```

2.1.4 El primer elemento de un arreglo es el “0” <- Cuidado con esto, es fuente constante de errores!!!

[8]:
```
# Pero si pido el elemento "1", veamos cuál es:

curso[1]
```

[8]:
```
'u'
```

[9]:
```
# También puedo pedir varios

curso[0:5]
```

[9]:
```
'Curso'
```

Ya que sería string[inicio:final:paso] <– Esta es una forma de notación muy general en Python Recordar que el índice inicial es el “0” y el final es el n-1. Si el final dice “4” entonces el último elemento es el “3” (ojo que igual hay cuatro elementos porque el primero tiene como índice al 0.

Si pido un elemento del string con un número negativo, significa que se empezará a contar los elementos desde el último, sin importar cuántos elementos tenga. Es una forma de acceder a los elementos finales sin tener que investigar cuántos elementos hay en total.

[10]:
```
curso[1:6:2]
```

[10]:
```
'us2'
```

[11]:
```
print('[0] =>', curso[0])
print('[0:1] =>', curso[0:1])
print('[0:2] =>', curso[0:2])
```

```
[0] => C
[0:1] => C
[0:2] => Cu
```

[12]:
```
print('[-1] =>', curso[-1])
print('[-2] =>', curso[-2])
print('[-1:-3:-1] =>', curso[-1:-5:-1])
```

```
[-1] => 2
[-2] => 2
[-1:-3:-1] => 2202
```

[13]:
```
# Y algo raro e impensado pero que tiene su lógica. Note que aplico
# la selección directamente al string. No hay variable

print('[::-1] =>', 'Curso'[::-1])
```

```
[::-1] => osruC
```

Se pueden incluso usar caracteres UNICODE

[14]:
```
strings = "Esto es Python"
char = "C"
multilinea_str = """Este es un string multilínea con más
de una línea de código."""
unicode = 'Yo \u2665 la cursada de Computación'
crudo_str = r"Este está crudo  \n string" # <-- prestar atención al r
                                          # antes del "
crudo_str2 = "Este está crudo  \n string"

# Imprimo el texto directo

print(strings)
print(char)
print("") # <---renglón en blanco

# En este caso tiene varios renglones al indicarse la asignación en
# la variable, pero se unen al asignarse

print(multilinea_str)
print("")

# Se pueden escribir caracteres UNICODE
#print(unicode)
```

```
#print("")
# Nota: los cuales no puedo pasar a este apunte porque el Latex no los reconoce


# O hacer que el texto se haga literal (sin ejecutar el "\n" que haría que
# se genere otro renglón). Esto se activa por el "r" antes del texto en la
# asignación.

print(crudo_str)
print(crudo_str2)
```

```
Esto es Python
C

Este es un string multilínea con más
de una línea de código.

Este está crudo  \n string
Este está crudo
 string
```

2.1.5 Variables Lógicas (Booleanas)

Sólo pueden guardar un "Verdadero" (True) o un "Falso" (False). Van con la primera letra en mayúscula y sin los puntos que vimos en FORTRAN.

[15]:
```
al = True
bl = False
print(al,bl)
print(type(al),type(bl))
```

```
True False
<class 'bool'> <class 'bool'>
```

Si bien el resultado es True o False, desde el punto de vista numérico hay un "1" en caso de una variable verdadera y un "0" en caso de que sea falso el valor guardado. Es decir las variables en si tienen guardado un número 1 o un 0 en ellas. Es decir me pueden servir para hacer cuentas. Por ejemplo, si multiplico una variable con un número por una booleana, en esta última habrá un 1 si es verdadera (y por lo tanto multiplicará por ese 1) o un 0 si es falsa.

Veamos esto:

[16]:
```
print('Uso a1 y b1 de la celda anterior')
print('Si multiplico un número por un booleano y era verdadero:', al*8)
print('Si multiplico un número por un booleano y era falso    :' ,bl*8)
```

```
Uso a1 y b1 de la celda anterior
Si multiplico un número por un booleano y era verdadero: 8
Si multiplico un número por un booleano y era falso    : 0
```

2.1.6 Convirtiendo tipos de Variables

USO las siguientes funciones, por eso llevan ()

- float() –> convierte a flotantes
- int() –> convierte a enteros
- str() –> convierte a strings

[17]:
```
a = True
print(a)
print(float(a))
print(int(a))
print(str(a))
print(int(13.97))
```

```
True
1.0
1
True
13
```

---

3 Operaciones con las variables

3.1 Operaciones matemáticas básicas

- Son iguales a FORTRAN
- Asignan los resultados de la misma manera a una tercera variable
- Hay operaciones no númericas
- Están las muy básicas y se usan exactamente igual que en Fortran, salvo una: +,-,*,/,**

- Y otras no tan convencionales, por ejemplo: //,%
- Existen asignaciones incrementales (o decrementales) sobre la misma variable (aditivas, multiplicativas)

[18]:
```
a = 2
b = 2
c = a + b
d = a - b
e = a * b
f = a ** b
g = a / b

print(c, 'es', a, '+', b)
print(d, 'es', a, '-', b)
print(e, 'es', a, '*', b)
```

```
print(f, 'es', a, '**', b)
print(g, 'es', a, '/', b, '¿Qué notan aquí?')
```

```
4 es 2 + 2
0 es 2 - 2
4 es 2 * 2
4 es 2 ** 2
1.0 es 2 / 2 ¿Qué notan aquí?
```

3.1.1 Las divisiones entre enteros generan un número real!!!!!

Esto es diferente a Fortran e incluso a las versiones anteriores de Python, incluyendo Python 2 Esta situación provocó un quiebre muy fuerte entre python 2 y 3. Provocando que mucha gente conserve simultáneamente la dos versiones por la incompatibilidad del software.

[19]:
```
# ejemplo

124231/1134
```

[19]:
```
109.55114638447972
```

4 Asignaciones múltiples

4.0.1 Se pueden hacer asignaciones múltiples

Es decir, que en una sentencia se asignan varios resultados.

El orden es determinante para entender a que variable le llega cada dato.

[20]:
```
x,y = 25.4, 54.67788
print(x,y)

z1,z2= x*24.5,y*4567
print(z1,z2)
```

```
25.4 54.67788
622.3 249713.87796
```

---

5 Operaciones no convencionales

[21]:
```
# ojo que existen estas operaciones. Esta es el resultado en
# enteros (floor division)

17//3
```

[21]: 5

[22]:
```
# y esta es el resto de la división. La función módulo en Fortran
17%3
```

[22]: 2

[23]:
```
# Las operaciones incrementales son válidas en Python, pero ojo no lo
# son las estilo C tipo i++

i=1

i+=10  #suma 10 a i  -> i=i+10
print(i)

i-=2  #resto 2 a i
print(i)

i*=4  #multiplico por 4 a i
print(i)

i/=5  #divido por 5 a i
print(i)
```

```
11
9
36
7.2
```

[24]:
```
# operaciones con strings (textos)
# Puedo sumar dos srings, pero quizás no es exactamente lo que uno espera
# Aunque tiene su lógica

a = 'Hola, estoy en '
b = 'la clase de computación'
x = a + b
print(x)
```

```
Hola, estoy en la clase de computación
```

---

6 Precisión de los cálculos

Es diferente a Fortran, los reales son siempre doble precisión (real*8) y los enteros tienen precisión arbitraria.

Recordar que las variables en “Precisión doble” dependen del hardware pero en general los números válidos están entre 10^-308 y 10^308, con 16 dígitos significativos

[25]:
```
numero=3242525225
z = numero**8
print(z)
```

```
122198790302863069638607705589529739839448039955553855075819700196838378906 25
```

[26]:
```
a=1
b=100000000000000000000
print(a/b)
```

```
1e-21
```

[27]:
```
# Recordar que puedo preguntar de que tipo son las variables
print(type(2))
print(type(2.3))

# Además puedo cortar (truncar) y redondear los números.

print(int(0.8)) # truncando
print(round(0.88766)) # ahora el más cercano pero que sea entero.

# Entendiendo el redondeo, en este caso se busca el par más cercano.

print(round(0.3))
print(round(1.5))
```

```
<class 'int'>
<class 'float'>
0
1
0
2
```

---

7 Números complejos

Hay que recordar el concepto de objetos, ahora lo veremos en el caso de números complejos, pero esto se extiende a todos los tipos de variables. En Python la parte imaginaria se indica con la letra j, no con i como es la convención normal que se usa en matemática.

Por lo cual un número complejo sería el siguiente:

[28]:
```
a = 1.5 + 0.5j
```

Puedo hacer cuentas conservando las propiedades de los números complejos

[29]:
```
a**2
```

[29]:
```
(2+1.5j)
```

Pero también puedo preguntar por la parte real e imaginaria por separado.

Ya que son atributos del Objeto número complejo

[30]:
```
a.real
```

[30]:
```
1.5
```

[31]:
```
a.imag
```

[31]:
```
0.5
```

Pero como también es un *Objeto* hay funciones asociadas en este. Hay que recordar que a diferencia de los atributos que ya están anotados, las funciones se calculan cuando son llamadas.

[32]:
```
a.conjugate() # En este caso la función requiere los ()
```

[32]:
```
(1.5-0.5j)
```

[33]:
```
a*a.conjugate()
```

[33]:
```
(2.5+0j)
```

---

8 Operaciones Booleanas

[34]:
```
5 < 7
```

[34]:
```
True
```

[35]:
```
a = 4
b = 9
```

[36]:
```
b < a  #  ¿Es cierto?
```

[36]:
```
False
```

[37]:
```
c = 1
```

[38]:
```
c < a < b        # Sería ver si 2 < 5 < 7 es cierto
```

[38]:
```
True
```

Los operados booleanos (o sea las operaciones lógicas) son **and, or, not** pero hay cosas como “is” (es un “and” literal)

[39]:
```
a < b and b < c
```

[39]: False

[40]:
```
resultado = a < 8
print(resultado, type(resultado))
```

```
True <class 'bool'>
```

[41]:
```
print(resultado)
print(not resultado)
```

```
True
False
```

[42]:
```
not resultado is True
```

[42]: False

También tenemos los comandos **is, is not**. El comando **is** también sirve para preguntar si el objeto es el mismo.

[43]:
```
a = 7
b = 7

print(a==b)
print(a is b)
```

```
True
True
```

[44]:
```
a = 22.4
resultado=22.4

print(resultado is a)
print(resultado==a)
```

```
False
True
```

9 Operaciones con Textos/Strings

[45]:
```
a = "Esto es un    string"
```

[46]:
```
# Pregunto su tamaño con una función que existe en Python

len(a)
```

[46]: 20

La variable **a** tal como la hemos creado es ahora un **Objeto** de la **Clase** string (textos). Puedo preguntar cuáles son las funciones o métodos de esa clase con la orden **dir()**

[47]:
```
print(dir(a))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize',
'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Los strings son objetos (como todo en python).
Recordar que hay métodos (o sea funciones) que están definidas en los objetos:

Tenemos dos strings B y C ¿Qué pasa si hacemos la operación B+C?

[48]:
```
B = "Primera Parte "
C = "Segunda Parte "

print(B+C)
```

```
Primera Parte Segunda Parte
```

Como se puede ver ambos strings se concatenan uno detrás del otro siguiendo el orden de la operación.

¿Y si hacemos D = B*2 ?

[49]:
```
D = B*2

print(D)
```

```
Primera Parte Primera Parte
```

Como generalización de la suma que vimos antes, se concatenan el mismo string la cantidad de veces que lo multiplico

9.0.1 Métodos del objeto strings

Veamos alguna de las funciones que tengo en la clase "string":

[50]:
```
a.upper()
```

[50]:
```
'ESTO ES UN    STRING'
```

[51]:
```
a.title()
```

[51]:
```
'Esto Es Un    String'
```

[52]:
```
a.split()
```

[52]:
```
['Esto', 'es', 'un', 'string']
```

Esto que se generó se llama **lista**, aunque parece un vector, no lo es exactamente y veremos las "listas" en la clase que viene.

Veremos que las listas tienen elementos y estos están numerados. El primer elemento está numerado como "0".

[53]:
```
a.split()[1]
```

[53]:
```
'es'
```

[54]:
```
a = "Este es un string.   Con muchas sentencias."
```

Lo puedo cortar eligiendo el caracter, por ejemplo, pruebo con el "."

[55]:
```
a.split('.')
```

[55]:
```
['Este es un string', '   Con muchas sentencias', '']
```

Por lo que obtuve una lista de sólo dos elementos.

[56]:
```
a.split('.')[1].strip()
# Defino el caracter para el split .
# Strip saca los blancos que están demás. El default es el blanco, puede␣
 ↪cambiarse a otro caracter
```

[56]:
```
'Con muchas sentencias'
```

[57]:
```
b = '   <-muchos espacios'
b.strip()
```

[57]:
```
'<-muchos espacios'
```

[58]:
```
a = 'tru'
b = 'la'
print(' '.join((a,b,b)))
print('-'.join((a,b,b)))
print(''.join((a,b,b)))
print(' '.join((a,b,b)).split())
print(' & '.join((a,b,b)) + 'lo')
```

```
tru la la
tru-la-la
trulala
['tru', 'la', 'la']
tru & la & lalo
```

Podemos hacer muchas cosas más con los strings. ¿Cómo se que cosas y cuáles son sus comandos?

Como siempre hay que buscar y leer el manual.

10 Ingresos de datos por teclado

10.0.1 Se utiliza la función input()

[59]:
```
base = float(input("base: "))
altura = float(input("altura: "))

area=base*altura/2

print("El área es=",area)
```

```
base: 23.4
altura: 53.45
El área es= 625.365
```