Python - Tipos de datos

Tipos de datos

El funcionamiento de los programas Python depende de los datos que maneja.
Todos los valores de datos en Python son **objetos**, y cada objeto, o valor, tiene "un tipo".
Tener en cuenta que, cada tipo de objeto determina qué operaciones va a soportar el objeto y, por lo tanto, qué operaciones se van a poder realizar con esos valores de los datos, qué atributos tiene y si va a poder ser "mutable" o no.
En Python también existe un tipo de dato **"objeto"** que acepta cualquier objeto como argumento y devuelve el tipo de objeto a el incorporado.
Por último, tener en cuenta que, en Python, también se han incorporado a tipos de datos: números, cadenas, tuplas, listas y diccionarios. Y, si estos no llegan, también se pueden crear tipos definidos por el usuario (**clases**).

Sumario

- 1 Números
  - 1.1 Números enteros
  - 1.2 Números de punto flotante
  - 1.3 Números complejos
- 2 Secuencias
  - 2.1 Tipos de datos "iterables"
    - 2.1.1 Strings
    - 2.1.2 Tuplas
    - 2.1.3 Listas
    - 2.1.4 Diccionarios
  - 2.2 Operaciones y funciones integradas para objetos iterables
    - 2.2.1 Operaciones comunes para los objetos iterables
    - 2.2.2 Funciones integradas que implican o devuelven objetos iterables
  - 2.3 Los conjuntos matemáticos *set* y *frozenset*
    - 2.3.1 set
    - 2.3.2 Los conjuntos inmutables
- 3 Problemas

Números

En Python los objetos **tipo número** soporta **enteros** (normales y largos), **números de punto flotante** y **números complejos**.

- Todos los números en Python son **"objetos inmutables"**, esto quiere decir que siempre que se realice una operación con un número el resultado será otro objeto número distinto.
- Las operaciones con números son "operaciones aritméticas".

Números enteros

- Un número entero puede ser decimal, octal o hexadecimal.

```
1, 23, 3493          # Decimales
01, 027, 06645       # Octales
0x1, 0x17, 0xDA5     # Hexadecimales
```

- En Python no es necesario distinguir entre enteros simples y enteros largos, pues él ya realiza el cambio cuando es necesario.

  Así y todo, sí podemos indicar con la letra **L** que va a ser un entero largo:

```
1L, 23L, 3493L          # Decimales largos
01L, 027L, 06645L       # Octales largos
0x1L, 0x17L, 0xDA5L     # Hexadecimales largos
```

Números de punto flotante

- En Python los decimales se indican con el dígito **.** (punto), una parte exponencial (con **e** o **E**) o ambos.

```
0., 0.0, .0
1.0, 1.e0, 1.0e0
```

Números complejos

- Un número complejo está realizado por dos números decimales, uno para la parte real y otro para la parte imaginaria.

  Podremos acceder a cada una de esas partes por separado: **z.real** y **z.imag**.
  Se especifica la parte imaginaria con la letra **j** o **J**.

```
0j, 0.j, 0.0j, .0j
1j, 1.j, 1.0j, 1e0j, 1.e0j, 1.0e0j
```

Secuencias

Una secuencia es un contenedor de artículos ordenados e indexados por enteros no negativos.

En Python tenemos: ***strings*** (simple y Unicode), **tuplas**, **listas** y **sets**.
Se tiene la posibilidad de crear nuevos tipos de secuencias.

Tipos de datos "iterables"

- El concepto de Python que generaliza la idea de "secuencia" es el concepto de "iterable".

  Todas las secuencias son iterables, pero existen más iterables, como las "listas", por ejemplo.
  Cuando hablamos de iterables nos referimos a "iterables acotados" (que tienen un principio y un final).

Strings

- Un objeto *string* (simple o Unicode) es una secuencia de caracteres utilizado para guardar y representar textos.

  Los *strings* en Python son inmutables.
  Los objetos *string* nos van a proporcionar muchos métodos.

- Para definir un *string* utilizaremos comillas simples o dobles:

```
'Este es un string'
"Este también es un string"
```

- Se utilizará *backslash* para escapar una comilla:

```
'Este es un \'string\''
"Este también es un 'string'"
```

- Para expandir el *string* a dos líneas utilizaremos también *backslash*:

```
"Este es un texto más largo\
que ocupa dos líneas"         # No pondremos el comentario en la primera línea
```

- Otro modo de crear un *string* de múltiples líneas es utilizando "tres comillas" (simples ó dobles, da igual).

```
"""Este sí que es
un texto muy
muy largo""" # Los comentarios siempre en la última línea
```

  El único carácter que no puede pertenecer a un *string*, así definido con tres comillas, es un *backslash* no escapado.

- Las secuencias de escape para variables *string* son las siguientes:

Secuencia	Significado
\<**nueva línea**>	El final de línea se ignora
\\	Backslash
\'	Comilla Simple
\"	Comilla Doble
n	Nueva línea
r	Retorno de carro
t	Tabulador

- Una variante del tipo **string** son las **raw string**, son utilizadas para que no realice su función el símbolo **backlash**. Para definir una **raw string** hay que colocar delante de la primera comilla una letra *r* o *R*.

- Formatear un string automáticamente con el método .format()

- Métodos de los objetos string en Python.

Tuplas

Una *tupla* es una secuencia de items **ordenada e inmutable**.
Los items de una tupla pueden ser objetos de cualquier tipo.
Para especificar una tupla, lo hacemos con los elementos separados por comas dentro de paréntesis.

- Una *tupla* con únicamente dos elementos es denominada ***par***.
- Para crear una *tupla* con un único elemento (*singleton*), se añade una coma al final de la expresión.
- Para definir una tupla vacía, se emplean unos paréntesis vacíos.

```
(100, 200, 300)    # Tupla con tres elementos
(3.14,)    # Tupla con un elemento
()    # Tupla vacía (los paréntesis NO son opcionales)
```

También se puede crear una tupla del siguiente modo: **saludo = tuple("hola")**

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
saludo = tuple("hola")
for letra in saludo:
    print ('%s' % letra)
# Vemos que, de este modo se crea una tupla igual a:
# saludo = ('h', 'o', 'l', 'a')
```

- Con **tuple()**, sin argumentos, creamos una tupla vacía.

Listas

- Una lista es una secuencia ordenada de elementos **mutable**.
- Los items de una lista pueden ser objetos de distintos tipos.
- Para especificar una lista se indican los elementos separados por comas en el interior de corchetes.
- Para denotar una lista vacía se emplean dos corchetes vacíos.

```
[42, 3.14, 'hola', 'casa']    # Lista con tres elementos
[100]    # Lista con un elemento
[]    # Lista vacía
```

- También podemos crear una lista del siguiente modo: ***list("hola")***

  que sería como crear una lista así: ['h', 'o', 'l', 'a']

- Podemos crear una lista vacía así: **list()**
- Podemos acceder a datos del siguiente modo:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

# Definimos una lista de 5 elementos
lista = [42, 3.14, 'hola', 25, 'fin']

# Ver todos los elementos de la lista
print ("Veamos los elementos de la lista:")
print (lista)

# Calculamos la longitud de la lista:
print ("\nLa longitud de la lista es de ", len(lista), "elementos")

# Las listas comienzan con el elemento "0"
print ("\nEl Primer elemento: %s" % lista[0])

# Con los índices negativos accedemos a los elementos al revés
print ("\nEl Último elemento: %s" % lista[-1])

# Para acceder a porciones de la lista:
# El primer índice de la porción especifica el primer elemento que se desea obtener,
# y el segundo índice especifica el primer elemento que no se desea obtener
print ("\nElementos segundo, tercero y cuarto: ", lista[1:4])

# append añade un elemento al final de la lista.
lista.append('nuevoFin')
print ("\nAñadimos un elemento final de la lista:")
print (lista)

# insert inserta un elemento en una lista.
# El argumento numérico es el índice del primer elemento que será desplazado de su posición.
lista.insert(2,'nuevo2')
print ("\nAñadimos un elemento en la posición 2:")
print (lista)

# extend concatena listas.
lista2 = [23, 45, 'lista2']
lista.extend(lista2)
print ("\nConcatenamos las dos listas: ")
print (lista)

# Borrar un elemento de la lista:
lista.remove(23)
print ("\nEliminamos el elemento 23")
print (lista)

# Eliminar un elemento de la lista por el índice:
del lista[1]
print ("\nEliminamos el elemento 1 de la lista:")
print (lista)

# Para eliminar el último elemento de la lista y verlo:
eliminado = lista.pop()
print ("\nSe eliminó el que estaba de último: ", eliminado)
print ("\nAhora la lista:")
print (lista)
```

Diccionarios

Los diccionarios de Python son una lista de consulta de términos de los cuales se proporcionan valores asociados.
En Python, un diccionario es una colección no-ordenada de valores que son accedidos a traves de una clave. Es decir, en lugar de acceder a la información mediante el índice numérico, como es el caso de las listas y tuplas, es posible acceder a los valores a través de sus claves, que pueden ser de diversos tipo.
Las claves son únicas dentro de un diccionario, es decir que no puede haber un diccionario que tenga dos veces la misma clave, si se asigna un valor a una clave ya existente, se reemplaza el valor anterior.
No hay una forma directa de acceder a una clave a través de su valor, y nada impide que un mismo valor se encuentre asignado a distintas claves

La informacion almacenada en los diccionarios, no tiene un orden particular. Ni por clave ni por valor, ni tampoco por el orden en que han sido agregados al diccionario.
Cualquier variable de tipo inmutable, puede ser clave de un diccionario: cadenas, enteros, tuplas (con valores inmutables en sus miembros), etc. No hay restricciones para los valores que el diccionario puede contener, cualquier tipo puede ser el valor: listas, cadenas, tuplas, otros diccionarios, objetos, etc.

- Veamos un ejemplo sacado de la web jarroba.com

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

# Definir una variable diccionario
futbolistas = dict()

futbolistas = {
    1: "Casillas", 3: "Piqué",
    5: "Puyol", 6: "Iniesta",
    7: "Villa", 8: "Xavi Hernández",
    9: "Torres", 11: "Capdevila",
    14: "Xavi Alonso", 15: "Ramos",
    16: "Busquets"
}
# Veamos el identificador inicial del diccionario
print ("Identificador del diccionario : {}\n".format(id(futbolistas)))

# Recorrer el diccionario, imprimiendo clave - valor
print ("Vemos que los elementos no van \"ordenados\":")
for k, v in futbolistas.items():
    print ("{} --> {}".format(k,v))

# Nº de elementos que tiene un diccionario
numElemen = len(futbolistas)
print ("\nEl número de futbolistas es de {}".format(numElemen))

# Imprimir las claves que tiene un diccionario
keys = futbolistas.keys();
print ("\nLas claves de nuestro diccionario son : {}".format(keys))

# Imprimir los valores que tiene un diccionario
values = futbolistas.values();
print ("\nLos valores de nuestro diccionario son : {}".format(values))

# Obtener el valor de un elemento dada su clave
elem = futbolistas.get(6)
print ("\nEl futbolista con clave 6 es {}".format(elem))

# Insertamos un elemento en el diccionario
## si la clave ya existe, el elemento NO se inserta
elem2 = futbolistas.setdefault(10, 'Cesc')
print ("\nInsertamos el elemento con clave 10 y valor Cesc")
print ("Ahora el identificador del diccionario : {}".format(id(futbolistas)))
print ("Ahora el diccionario queda : {}".format(futbolistas))

# Añadimos, de otro modo, un elemento al diccionario
## si la clave ya existe, el valor se cambia por este nuevo
futbolistas[22] = 'Navas'
print ("\nAñadimos un nuevo elemento, ahora el diccionario queda: ")
print (futbolistas)
print ("Ahora el identificador del diccionario : {}".format(id(futbolistas)))

# Eliminamos un elemento del diccionario dada su clave
futbolistas.pop(22)
print ("\nEliminamos el elemento con clave 22")
print ("Ahora el diccionario queda: {}".format(futbolistas))
print ("Ahora el identificador del diccionario : {}".format(id(futbolistas)))

# Hacemos una copia del diccionario
futbolistas_copia = futbolistas.copy()
print ("\nLa copia del diccionario tiene los valores:")
print (futbolistas_copia)
print ("El identificador de la copia del diccionario : {}".format(id(futbolistas_copia)))
```

```
# Borramos los elementos de un diccionario
futbolistas_copia.clear()
print ("\nVaciamos el diccionario nuevo creado, ahora los valores en el: {}".format(futbolistas_copia))

# Creamos un diccionario a partir de una lista de claves
keys = ['nombre', 'apellidos', 'edad']
datos_usuario = dict.fromkeys(keys, 'null')
print ("\nCreamos un diccionario a partir de la lista de claves:")
print (keys)
print ("y con valor 'null'")
print ("Así el diccionario queda: {}".format(datos_usuario))

# Comprobamos si existe o no una clave en un diccionario
if 2 in futbolistas:
    print ("\nEl futbolista con la clave 2 existe en el diccionario.")
else:
    print ("\nEl futbolista con la clave 2 NO existe en el diccionario.")

if 8 in futbolistas:
    print ("\nEl futbolista con la clave 8 existe en el diccionario.")
else:
    print ("\nEl futbolista con la clave 8 NO existe en el diccionario.")

# Devolvemos los elementos del diccionario en una tupla
tupla = futbolistas.items()
print ("\nEl diccionario convertido en tupla queda así:")
print (tupla)

# Unimos dos diccionarios existentes
suplentes = {
    4:'Marchena', 12:'Valdes', 13:'Mata',
    17:'Arbeloa', 19:'Llorente', 20:'Javi Martinez',
    21:'Silva', 23:'Reina'
}

print ("\nUnimos el diccionario: ")
print (futbolistas)
futbolistas.update(suplentes) # Aquí hacemos la unión de los diccionarios
print ("con el diccionario:")
print (suplentes)
print ("siendo el resultado:")
print (futbolistas)
```

Operaciones y funciones integradas para objetos iterables

Python poseen funciones integradas y operaciones comunes que implican o devuelven los objetos iterables.

Operaciones comunes para los objetos iterables

Las principales operaciones comunes para los objetos iterables son el **desempaquetado**, la **prueba de pertenencia**y la **iteración**.

- **Desempaquetado**

  Los objetos iterables admiten el desempaquetado.

```
>>> a, b, c = (1, 2, 3) # Los paréntesis no son necesarios
>>> a
1
>>> a, b, c = 'sol'
>>> a
's'
>>> a, b, c = {'nombre': 'Manuel', 'apellido': 'Rodríguez', 'edad': '40'} # Diccionario
>>> a
'nombre'
```

  También se puede realizar el **desempaquetado extendido** (*extended iterable unpacking* - PEP-3132), que se diferencia del clásico por la presencia de una etiqueta marcada con un asterisco.

```
>>> *a, b, c = 'python'
>>> a
```

```
['p', 'y', 't', 'h']
>>> a, *b, c = 'python'
>>> a
'p'
>>> b
['y', 't', 'h', 'o']
>>> c
'n'
>>> a, *b, c, d = [1, 2, 3]
>>> a
1
>>> b
[]
>>> c
2
>>> d
3
```

- **Prueba de pertenencia**

  La palabra clave *in* permite comprobar la pertenencia de un elemento a un objeto iterable.
  Y, con la palabra clave *not* combinada con la *in* podemos comprobar la no pertenencia.

```
>>> 'yth' in 'python'
True
>>> 'a' in ['a', 'b', 'c']
True
>>> 'nombre' in {'nombre': 'Manuel', 'edad': 40}
True
>>> 'Manuel' in {'nombre': 'Manuel', 'edad': 40}
False
>>> 'apellidos' not in {'nombre': 'Manuel', 'edad': 40}
True
```

- **Iteración**

  La principal característica de los objetos iterables, es que se puede acceder a sus elementos, uno a uno, mediante la instrucción for:

```
>>> for letra in 'python':
...     print(letra)
...
p
y
t
h
o
n
```

Funciones integradas que implican o devuelven objetos iterables

- Recuerda que la función bool(x) devuelve *True* si, *x* es *False*, *0* o se omite.
- La función all(iterable) devuelve *True* si, para cada *x* que pertenece a *iterable*, la función *bool(x)* devuelve *True*, o si el objeto iterable está vacío.

```
>>> all([0, 1, 2, 3, 4])
False
>>> all([1, 2, 3, 4])
True
>>> all([])
True
>>> all([0, 0, 0])
False
```

- La función any(iterable) devuelve *True* si hay al menos un elemento *x* del objeto iterable pasado como argumento para el cual *bool(x)* devuelve *True*.

```
>>> any([0, 1, 2, 3, 4])
True
>>> any([1, 2, 3, 4])
```

```
True
>>> any([])
False
>>> any([0, 0, 0])
False
```

- Las funciones max(iterable) y min(iterable) devuelven el elemento máximo y mínimo de un objeto iterable.

    ◊ Admiten ambas un argumento opcional *key* que permite aplicar una regla. Así, si ponemos *key=len* nos devuelve el objeto más largo, o más corto, de los iterables pertenecientes a ese iterable.
    ◊ Si ponemos *key=abs* nos devuelve el máximo, o el mínimo, en valor absoluto.
    ◊ También admiten el argumento *default* que nos permite indicar cuál será la salida si el iterable está vacío.

- La función sum(iterable) calcula la suma de los elementos de un objeto iterable.
- La función sorted(iterable) devuelve una lista con los elementos del objeto iterable dispuestos ordenadamente.

    ◊ Podemos utilizar esta función *sorted()* para iterar en las claves de un diccionario de forma ordenada:

```
>>> datos = {'nombre': 'Manuel', 'apellidos': 'Vieites', 'edad': 40}
>>> for k in sorted(datos):
...     print(k, datos[k], sep=' -> ')
...
apellidos -> Vieites
edad -> 40
nombre -> Manuel
```

    ◊ La función *sorted()* tiene el argumento opcional *reverse* que devuelve el iterable de menor a mayor.

- La función zip() toma como argumentos un número arbitrario de objetos iterables y devuelve un iterador del siguiente modo:

```
>>> a = [10, 15, 22, 30, 45]
>>> b = 'Python'
>>> z = zip(a, b)
>>> type(a)
<class 'list'>
>>> type(b)
<class 'str'>
>>> type(z)
<class 'zip'>
>>> for item in z:
...     print(item)
...
(10, 'P')
(15, 'y')
(22, 't')
(30, 'h')
(45, 'o')
>>>
```

- La función map() toma como argumentos una función y uno o más objetos iterables y devuelve un iterador.

```
>>> def producto(x, y):
...     return x * y
...
>>> for i in map(producto, (2, 3, 4), (10, 20)):
...     print(i)
...
20
60
>>>
```

- La función filter() devuelve un iterable como resultado de otro iterable dado como parámetro y una función para los cuales esta función devuelve True. Con un ejemplo lo entenderemos perfectamente.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

# Función filtra-Vocales
```

```
def filtraVocales(texto):
    vocales = ['a', 'e', 'i', 'o', 'u']
    if(texto.lower() in vocales):
        return True
    else:
        return False
#--------------------------

textoPrueba = 'Este es un texto con vocales'

vocalesFiltradas = filter(filtraVocales, textoPrueba)

# Mostramos la salida
print('Texto de trabajo: ')
print('\t' + textoPrueba)
print('Vocales existentes: ')
for v in vocalesFiltradas:
    print('\t' + v)

## Salida:
# Texto de trabajo:
#         Este es un texto con vocales
# Vocales existentes:
#         E
#         e
#         e
#         u
#         e
#         o
#         o
#         o
#         a
#         e
```

Los conjuntos matemáticos *set* y *frozenset*

En Python, los tipos integrados *set* y *frozenset* representan **conjuntos matématicos**. La diferencia entre ambos es que las instancias de la clase *set* son objetos mutables y las de la clase *frozenset* son inmutables.

set

Las instancias de tipo set son contenedores mutables pero, ojo, de objetos no ordenados, únicos e inmutables, y tienen las mismas propiedades que los conjuntos matemáticos.

```
#Definimos un objeto 'set'
>>> s = set([1, 'hola', 2, ('a', 'b')])
#Son todos elementos inmutables: enteros, strings, tuplas
#Le añadimos un nuevo elemento (un entero)
>>> s.add(4)
#Mostramos por pantalla el contenido del objeto
>>> s
{1, 2, 4, 'hola', ('a', 'b')}
#Intentamos añadirle una lista, que es un objeto 'mutable'
>>> s.add([1, 2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
#Intentamos añadirle un elemento que ya existe
>>> s.add(1)
#Vemos que no lo añade, no puede tener elementos repetidos
>>> s
{1, 2, 4, 'hola', ('a', 'b')}
#Podemos calcular su tamaño con la función integrada len()
>>> len(s)
5
>>>
```

Los objetos *set* se comportan exactamente como los conjuntos matemáticos.

- Intersección: *s1 & s2*

- Unión: *s1 | s2*
- Diferencia: *s1 - s2*
- Diferencia asimétrica: *s1 ^ s2*

Los conjuntos inmutables

Hemos dicho que los conjuntos son objetos mutables que pueden tener como elementos solo objetos inmutables. Esto significa que un *set* no puede contener en su interior otro *set*. Por ese motivo se crearon los conjuntos *frozenset*, que son conjuntos que pueden contener objetos mutables: Eso sí, las instancias de la clase *frozenset* son objetos inmutables. Podemos realizar operaciones entre instancias *set* y *frozenset*, que darían como resultado un *frozenset*.

Problemas

1.- Escribir un programa Python que declare, usando nombres descriptivos, variables de Python que representen: meses del año, segundos en un minuto, una posición de memoria de un ordenador, la constante Pi y un valor lógico. Asignarles un valor y que se muestren por pantalla.

Resolución.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
from math import pi
# Declaración de variables
mes_del_anyo = "Enero" #Mes del año
segundos = 60 # Segundos en un minuto
posicion_memoria = 65536 # Posición de memoria
valor_logico = True # Valor lógico True - False

# Mostrar por pantalla
print ("Estamos en el mes del año " + mes_del_anyo)
print ("Pasaron ya  {} de este minuto".format(segundos))
print ("Ese dato está en la posición de memoria {}".format(posicion_memoria))
print ("El número Pi vale {:.2f}".format(pi))
if valor_logico:
    print ("Esa afirmación es CORRECTA")
else:
    print ("Esa afirmación es FALSA")
```

2.- Hacer un programa que pida el radio por pantalla y devuelva el perímetro, el área y el volumen utilizando dicho parámetro:

Nota: En la resolución de este problema se utiliza el módulo math.

Resolución.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

#Programa que pide un radio de un círculo o esfera
#Calcula:
### El perímetro : 2*Pi*Radio
### El área : Pi * Radio*Radio
### El volumen : 4 * Pi * Radio * Radio * Radio / 3

#Importar módulos y funciones necesarias
#import math #Importa todo el módulo math
##Para llamar a pi : math.pi
##Vamos a importar sólo la constante pi
from math import pi, pow
from os import system

##Definición de variables
radio = float(0)
perimetro = float(0)
area = float(0)
volumen = float(0)
salida = ""
opcion = ""

while(True):
    try:
```

```
        ##Borro la pantalla
        system('cls')
        ##Pedir datos al usuario
        radio = float(input("Introduce radio :").replace(",", "."))

        ##Hacemos los cálculos
        perimetro = 2 * pi * radio
        area = pi * pow(radio, 2)
        volumen = 4 * pi * pow(radio, 3) / 3

        #Configuramos la salida
        salida = "\tEl círculo de radio {} tiene de perímetro {}\n".format(radio, round(perimetro, 2))
        salida += "\tTiene de área : {}\n".format(round(area, 2))
        salida += "\tY si fuese una esfera, de volumen : {}".format(round(volumen, 2))

        #Mostramos por pantalla
        print(salida)
    except:
        print("\t¡¡¡El radio debe ser un número!!!")
    finally:
        print("\n\n")
        opcion = input("¿Quieres volver a empezar(s/n)? ")
        if opcion.lower() != "s":
            break
        else: #En python cualquier if debería ir acompañado de else
            pass #Si no va a hacer nada
```

3.- Diseñar un programa que pida la longitud de los dos lados distintos de un rectángulo (en metros) y, con ese dato, calcule el valor de su perímetro (metros) y el de su área (metros cuadrados)

4.- El área A de un triángulo e puede calcular a partir del valor de dos de sus lados, a y b, y del ángulo ø que estos forman entre sí con la fórmula: A = 1/2 * a * b * sin(ø). Diseña un programa que pida el valor de los dos lados (en metros), el valor del ángulo que estos forman (en grados), y muestre el valor del área.

> [Ten en cuenta que la función **sin** de Python trabaja en radianes, así que el ángulo que se lea en grados hay que pasarlo a radianes sabiendo que ? radianes son 180 grados].

5.- Escribir un programa que calcule la ecuación del espacio recorrido por un objeto que se mueve a una velocidad constante (Espacio [m] = Velocidad [m/s] * Tiempo [s]).

6.- Escribir un programa que calcule las raíces de la ecuación cuadrática: a*x2 + b*x + c = 0 y las imprima por pantalla. Debe pedir por pantalla los coeficientes a, b y c.

7.- Escribir un programa que declare dos variables ***string*** con valores "Hola" y "alumno" y compruebe si son iguales, su longitud, cuál es su segundo carácter, concatene las dos variables e imprima el substring de 2 a 6 de las variables concatenadas.

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

# Definimos las variables
saludo = "Hola"
sujeto = "alumno"

# Comprobamos si son o no iguales
if saludo == sujeto:
    print ("Las variables son iguales\n")
else:
    print ("Las variables son distintas\n")

# Calculamos su longitud
print ("La cadena '{}' tiene una tamaño de {} caracteres".format(saludo, len(saludo)))
print ("La cadena '{}' tiene una tamaño de {} caracteres".format(sujeto, len(sujeto)))

# Mostramos su segundo carácter
print ("El segundo carácter de '{}' es '{}'".format(saludo, saludo[1]))
print ("El último carácter de '{}' es '{}'".format(sujeto, sujeto[-1]))

# Concatenar las dos variables e imprimir el substring de 2 a 6 de las variables concatenadas
print ("Substring 2 a 6 de las dos variables concatenadas: {}".format(str(saludo+sujeto)[1:5]))
```

-- Volver