Introducción a Python (Matemáticas 2. GII/I2ADE)

Preámbulo

Instalación en ordenadores personales

En el Moodle de la asignatura puedes encontrar un tutorial de como instalar el `miniconda y jupyter notebook` en tu ordenador personal.

Ordenadores de clase

En los ordenadores de clase todo el entorno está ya instalado y por lo tanto solamente hay que ejecutar `jupyper-notebook` en la consola para empezar a trabajar (ten en atención a la directoria donde el terminal se encuentra pues esa será la directoria base para el `jupyter`).

Librerías necesarias para esta asignatura

Para el desarrollo de las prácticas de esta asignatura es necesario utilizar un conjunto de librerías que es necesario importarlas tal y como se indica a continuación:

In [1]:
```
import math
import sympy
import scipy
import numpy as np
import matplotlib.pyplot as plt
```

De manera que:

Math es una librería primitivas matemáticas nativas de Python.

Sympy es una librería para facilitar la realización de cálculos matemáticos con variables sin necesidad de definir funciones nativas de Python.

Scipy es una librería con diversas funciones para matemáticas y ingeniería.

Numpy es la librería para realizar cálculos y manipulación numérica por excelencia.

Matplotlib es una librería para gráficos.

Introducción

Python es un lenguaje de programación **interpretado**, es decir, que no se compila, sino que es interpretado en el momento de ejecución. Por eso, es muy **flexible** y, al mismo tiempo, es muy **fácil cometer errores**. Aunque existen formas de capturar algunos errores pre-ejecución, estas formas son muy básicas y muchísimos errores no se detectan. Python está orientado a objetos y puede utilizarse para *scripting* de una forma más formal.

Así pues, el código escrito en Python puede ejecutarse (al menos) de tres formas:

1. Modo interpretativo (tipo terminal/símbolo del sistema): en este modo, todos los comandos son interpretados continuamente. En el caso de que se cierre la sesión, se pierden todo el código y los datos. Es bueno para probar algún comando rápidamente.
2. Modo fichero (modo interpretativo, ejecuta cada línea del fichero(s)): es la forma estándar de ejecutar código en Python.
3. Modo interactivo (Jupyter/IPython): es el formato utilizado en este fichero. A diferencia del anterior, el código se ejecuta bajo demanda del usuario y puede ejecutarse por partes.

Python es *"with wheels included"*, es decir, incluye muchas funcionalidades de base, aunque se suele decir que existen *librerías para todo*.

El código de Python es IDENTADO. Es decir, no existen símbolos ni palabras clave para marcar la terminación de una línea ni de un bloque (condicionales/bucles), sino que la identación es lo que define la "pertenencia" a las distintas estructuras, como veremos a lo largo de la asignatura.

Conceptos Básicos

Aritmética

In [2]:
```
1 + 2
```
Out[2]: 3

In [3]:
```
4 % 4  # resto de la división
```
Out[3]: 0

In [4]:
```
1 * 3
```
Out[4]: 3

In [5]:
```
1 / 2
```
Out[5]: 0.5

In [6]:
```
2**4  # potencia
```

Out[6]: 16

In [7]:
```
2 + 3 * 5 + 5  # precedencia de los operadores
```

Out[7]: 22

In [8]:
```
(2 + 3) * (5 + 5)  # cambios en la precedencia de los operadores
```

Out[8]: 50

In [9]:
```
# Redondeo hacia + infinito
math.ceil(2.354)
```

Out[9]: 3

In [10]:
```
# Redondeo hacia - infinito
math.floor(2.354)
```

Out[10]: 2

In [11]:
```
# Máximo común divisor
math.gcd(48, 60)
```

Out[11]: 12

In [18]:
```
# Redondeo hasta x decimales
print(round(3.564, 2))
print(round(3.564, 1))
```

```
3.56
3.6
```

Trigonometría

Para usar funciones trigonométricas es necesario usar las librerías comentadas anteriormente. Se debe **tener en atención que ciertas constantes no son equivalentes entre las distintas librerías**.

In [21]:
```
# EJEMPLO PARA PI
print(math.pi == np.pi)
print(math.pi == scipy.pi)
print(math.pi == sympy.pi) # sympy.pi no es un número
```

```
True
True
False
```

Constantes

In [22]:
```
# pi
math.pi
```

Out[22]: 3.141592653589793

In [23]:
```
# Euler/Napier
math.e
```

Out[23]: 2.718281828459045

In [24]:
```
# 2*pi
math.tau
```

Out[24]: 6.283185307179586

In [25]:
```
# infinito
math.inf
```

Out[25]: inf

In [26]:
```
# infinito
-math.inf
```

Out[26]: -inf

In [27]:
```
# no-número (not-a-number)
math.nan
```

Out[27]: nan

Conversión

In [28]:
```
# Radianes -> Grados
math.degrees(math.pi)
```

Out[28]: 180.0

In [29]:
```
# Grados -> Radianes
math.radians(180)
```

Out[29]: 3.141592653589793

Funciones Trigonométricas

In [30]:
```
# coseno -> resultado en RADIANES
math.cos(1)
```

Out[30]: 0.5403023058681398

In [31]:
```
# seno
math.sin(1)
```

Out[31]: 0.8414709848078965

In [32]:
```
# tangente
math.tan(1)
```

Out[32]: 1.5574077246549023

In [33]:
```
# arcocoseno -> en radianes, entre 0 y pi
math.acos(0.5)
```

Out[33]: 1.0471975511965979

In [34]:
```
# arcoseno -> en radianes, entre -pi/2 y pi/2
math.asin(0.5)
```

Out[34]: 0.5235987755982989

In [35]:
```
# arcotangente -> en radianes, entre -pi/2 y pi/2
math.atan(0.5)
```

Out[35]: 0.4636476090008061

Funciones Hiperbólicas

In [36]:
```
# arcocoseno hiperbolico
math.acosh(1)
```

Out[36]: 0.0

In [37]:
```
# arcoseno hiperbolico
math.asinh(1)
```

Out[37]: 0.881373587019543

In [38]:
```
# arcotangente hiperbolica
math.atanh(0.5)
```

Out[38]: 0.5493061443340549

In [39]:
```
# coseno hiperbolico
math.cosh(1)
```

Out[39]: 1.5430806348152437

In [40]:
```
# seno hiperbolico
math.sinh(1)
```

Out[40]: 1.1752011936438014

In [41]:
```
# tangente hiperbolica
math.tanh(1)
```

Out[41]: 0.7615941559557649

Potencias y logaritmos

In [42]:
```
# Exponencial
math.exp(1)
```

Out[42]: 2.718281828459045

In [43]:
```
# Logaritmo
# math.log(x[, base])
# Con 1 argumento tiene base "e"
# Con 2 argumentos se calcula como log(x)/log(base)
math.log(2)
```

Out[43]: 0.6931471805599453

In [44]:
```
math.log(2, 5)
```

Out[44]: 0.43067655807339306

In [45]:
```
# Logaritmo base 10
math.log10(2)
```

Out[45]: 0.3010299956639812

In [46]:
```
# Potencia
math.pow(3, 2)  # devuelve float
```

Out[46]: 9.0

In [47]:
```
# o
3**2 # devuelve int
```

Out[47]: 9

In [48]:
```
# Raíz cuadrada
math.sqrt(2)
```

Out[48]: 1.4142135623730951

Definición de variables

Los nombres de las variables no pueden empezar con caracteres especiales ni con números

In [49]:
```
name_of_var = 2
x = 2
y = 3
z = x + y
z
```

Out[49]: 5

Cadena de caracteres

Pueden escribirse entre comillas simples o comillas dobles y pueden incluir comillas simples cuando van entre comillas dobles.

In [50]:
```
"una frase"
"otra frase"
"otra's frase"
"HoLa SoY YO, el tRueno".lower()
```

Out[50]: 'hola soy yo, el trueno'

In [54]:
```
num = 12
name = "Sam"
print("Mi número es: {one}, y mi nombre es: {two}".format(one=num, two=nam
e))
```

```
Mi número es: 12, y mi nombre es: Sam
```

In [55]:
```
"Mi número es: {}, y mi nombre es: {}".format(num, name)
```

Out[55]: 'Mi número es: 12, y mi nombre es: Sam'

In [56]:
```
f"Mi número es: {num}, y mi nombre es: {name}"
```

Out[56]: 'Mi número es: 12, y mi nombre es: Sam'

Mostrar por pantalla

Cuando se usa Jupyter no es necesario utilizar el comando *print*

In [57]:
```
print("Hola")
```

```
Hola
```

In [58]:
```
print("Hola", "mundo")
```

```
Hola mundo
```

In [59]:
```
print("Hola", "mundo", sep=" :: ")
```

```
Hola :: mundo
```

Listas

In [60]:
```
[1, 2, 3]
```

Out[60]: [1, 2, 3]

In [61]:
```
["hi", 1, [1, 2]]
```

Out[61]: ['hi', 1, [1, 2]]

In [62]:
```
my_list = ["a", "b", "c"]
my_list.append("d")
my_list
```

Out[62]: ['a', 'b', 'c', 'd']

In [63]:
```
my_list[0]
```

Out[63]: 'a'

In [64]:
```
my_list[2]
```

Out[64]: 'c'

El operador `:` permite definir un rango de valores. Este operador suele utilizarse para definir los índices de los elementos de una lista que queremos obtener. Así, por ejemplo:

`0:3` define una lista que comprende desde el 0 hasta el 2, por lo que los elementos que nos devolverá será los correspondientes a las posiciones 0, 1 y 2(la posición 3 no se incluye).

`1:` va desde la posición 1 hasta el final

`:3` va desde la posición 0 hasta la 2

In [ ]:
```
my_list[1:]  # Extracto de la lista my_list['a', 'b', 'c', 'd']
```

Out[ ]: ['b', 'c', 'd']

In [ ]:
```
my_list[:2]
```

Out[ ]: ['a', 'b']

In [ ]:
```
my_list[2:4]
```

Out[ ]: ['c', 'd']

In [65]:
```
my_list[0] = "NEW"  # cambia el valor existente
my_list
```

Out[65]: ['NEW', 'b', 'c', 'd']

In [66]:
```
nest = [1, 2, 3, [4, 5, ["target"]]]  # listas de listas
nest[3]
```

Out[66]:
```
[4, 5, ['target']]
```

In [67]:
```
nest[3][2]  # acceso a elementos de listas de listas
```

Out[67]:
```
['target']
```

In [68]:
```
nest[3][2][0]  # acceso a elementos de listas de listas
```

Out[68]:
```
'target'
```

In [69]:
```
# los caracteres de las cadenas de caracteres se acceden como las listas
nest[3][2][0][3]
```

Out[69]:
```
'g'
```

In [70]:
```
list(range(5))
```

Out[70]:
```
[0, 1, 2, 3, 4]
```

Diccionarios

Definen un conjunto de elementos que tienen la estructura `"clave":"item"`

In [71]:
```
d = {"key1": "item1", "key2": "item2"}
d["key1"]
```

Out[71]:
```
'item1'
```

In [72]:
```
d.keys()
```

Out[72]:
```
dict_keys(['key1', 'key2'])
```

In [73]:
```
d.values()
```

Out[73]:
```
dict_values(['item1', 'item2'])
```

In [74]:
```
d = {"0001": {"nombre": "Pepe", "edad": 20, "ciudad": "Alicante", "gamer": True}}
d
```

Out[74]:
```
{'0001': {'nombre': 'Pepe', 'edad': 20, 'ciudad': 'Alicante', 'gamer': True}}
```

In [75]:
```
d["0001"]["edad"]
```

Out[75]:
```
20
```

In [76]:
```
"ciudad" in d["0001"]  # verifica si existe una clave
```

Out[76]:
```
True
```

In [77]:
```
d["0001"]["movil"]  # si una clave no existe, nos devuelve error
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[77], line 1
----> 1 d["0001"]["movil"]

KeyError: 'movil'
```

In [78]:
```
d["0001"].get("movil", "No existe")
```

Out[78]:
```
'No existe'
```

In [82]:
```
d["0001"].get("ciudad", "No existe")
```

Out[82]:
```
'No existe'
```

In [81]:
```
del d["0001"]["ciudad"]  # Borra item "ciudad"
d
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[81], line 1
----> 1 del d["0001"]["ciudad"]  # Borra item "ciudad"
      2 d

KeyError: 'ciudad'
```

Tuplas

Las tuplas tienen la particularidad de que no pueden modificarse, es decir, son inmutables.

In [83]:
```
t = (1, 2, 3)
t[0]
```

Out[83]:
```
1
```

In [84]:
```
t[0] = "nuevo"  # Error porque es inmutable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[84], line 1
----> 1 t[0] = "nuevo"

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

Conjuntos (Sets)

Son listados que no admiten elementos duplicados.

In [108]:
```
s = {1, 2, 3, 2, 1, 4}
s
```

Out[108]:
```
{1, 2, 3, 4}
```

In [109]:
```
s.add(5)
s
```

Out[109]:
```
{1, 2, 3, 4, 5}
```

In [110]:
```
s.add(1)
s
```

Out[110]:
```
{1, 2, 3, 4, 5}
```

In [111]:
```
s.pop()  # Devuelve el primer elemento del conjunto y lo BORRA
```

Out[111]:
```
1
```

In [112]:
```
s
```

Out[112]:
```
{2, 3, 4, 5}
```

In [113]:
```
del s[2]  # Error
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[113], line 1
----> 1 del s[2]

TypeError: 'set' object doesn't support item deletion
```

Condicionales

Hay que tener mucho cuidado con la indentación

In [ ]:
```
# Ejemplo básico
if 1 < 2:
    a = True
    print("Es verdadero")
else:
    a = False
    print("Es falso")
```

```
Es verdadero
```

In [ ]:
```
if (1 < 2) and (2 < 3):
    print("Ejemplo AND")
if (1 < 2) or (2 < 1):
    print("Ejemplo OR")
if not False:
    print("Ejemplo negación")
```

```
Ejemplo AND
Ejemplo OR
Ejemplo negación
```

In [ ]:
```
if 1 == 2:
    print("Primero")
elif 3 == 3:  # ejemplo else-if
    print("Segundo")
else:
    print("Tercero")
```

```
Segundo
```

Bucles

For

In [ ]:
```
seq = [1, 2, 3, 4, 5]
for item in seq:
    print(item)
    print(f"Su doble es: {item+item}")
```

```
1
Su doble es: 2
2
Su doble es: 4
3
Su doble es: 6
4
Su doble es: 8
5
Su doble es: 10
```

In [15]:
```
animales = ["gato", "perro", "pajaro"]
for k in animales:
  print(k)
```

```
gato
perro
pajaro
```

In [ ]:
```
# range(5) es lo mismo que range(0,5)
for i in range(5):
    print(i, f"y su doble es: {i*2}")
```

```
0 y su doble es: 0
1 y su doble es: 2
2 y su doble es: 4
3 y su doble es: 6
4 y su doble es: 8
```

In [ ]:
```
animales = ["gato", "perro", "pajaro"]
for idx, animal in enumerate(animales):
    print("#%d: %s" % (idx + 1, animal))
```

```
#1: gato
#2: perro
#3: pajaro
```

While

In [16]:
```
i = 0
while i < 3:
    print(f"#{i + 1}: {animales[i]}")
    i = i + 1
```

```
#1: gato
#2: perro
#3: pajaro
```

List comprehension

Una List comprehension consiste en un atajo para crear una lista a partir de un `for` sin que sea necesario instanciar una variable para contener una lista. Es también posible filtrar (con `if`) y encadenar varios `for` e `if` dentro.

In [ ]:
```
[item**2 for item in [1, 2, 3, 4]]  # crea una lista a partir de un bucle for
```

Out[ ]:
```
[1, 4, 9, 16]
```

In [ ]:
```
# crea una lista a partir de un bucle for teniendo en cuenta que se cumple una condición (if)
[item for item in [1, 2, 3, 4] if item % 2 == 0]
```

Out[ ]:
```
[2, 4]
```

In [19]:
```
# devuelve una lista con el texto
[f"#{idx + 1}: {animal}" for idx, animal in enumerate(["gato", "perro", "pajaro"])]
```

Out[19]:
```
['#1: gato', '#2: perro', '#3: pajaro']
```

In [20]:
```
# Multiples "if"
datos = [("gato", 89), ("perro", 45), ("gato", 92)]
[i for i in datos if i[0] == "gato" if i[1] >= 90]
```

Out[20]:
```
[('gato', 92)]
```

Detección de Tipos

En Python se puede detectar el tipo de cada variable. Esto porque en Python, una variable puede asumir cualquier tipo en cualquier momento.

In [ ]:
```
isinstance(1, int)
```

Out[ ]:
```
True
```

In [ ]:
```
isinstance(8.0, float)
```

Out[ ]: True

In [ ]:
```
isinstance(8.0, int)
```

Out[ ]: False

In [ ]:
```
isinstance("abc", str)
```

Out[ ]: True

Funciones

Las funciones son los componentes fundamentales de Python. Su sintaxis es la siguiente:

```
def nombre_de_la_funcion(argumentos):
  código
  código
  código
  return algo # esto es opcional
```

La identación es fundamental para que funcione correctamente. Los argumentos pueden tener valores por defecto para el caso en que no se pase ningún dato como argumento.

In [115]:
```
def signo(x):
    if x > 0:
        return "positivo"
    elif x < 0:
        return "negativo"
    else:
        return "zero"


for x in [-1, 0, 1]:
    print(signo(x))
```

```
negativo
zero
positivo
```

Desafio

Prueba la función `signo` pero con un caracter como argumento.

In [116]:
```
signo(">")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[116], line 1
----> 1 signo(">")

Cell In[115], line 2, in signo(x)
      1 def signo(x):
----> 2     if x > 0:
      3         return "positivo"
      4     elif x < 0:

TypeError: '>' not supported between instances of 'str' and 'int'
```

In [117]:
```
def hola(nombre="Ricardo", sonoro=False):
    if sonoro:
        # .upper() devuelve el texto en mayúsculas
        print(f"HOLA {nombre.upper()}")
    else:
        print(f"Hola {nombre}")


hola("Bob")
```

```
Hola Bob
```

In [118]:
```
hola("Fran", sonoro=True)
```

```
HOLA FRAN
```

In [119]:
```
hola("paco", True)
```

```
HOLA PACO
```

In [120]:
```
hola()
```

```
Hola Ricardo
```

In [121]:
```
hola(sonoro=True)
```

```
HOLA RICARDO
```

In [ ]:
```
# Una función que divide una lista de números en pares y impares
def par_impar(lista):
    par = []
    impar = []
    for i in lista:
        if i % 2:
            par.append(i)
        else:
            impar.append(i)
    return par, impar


par, impar = par_impar([1, 5, 6, 7, 85, 23, 56, 92])
print(par)
print(impar)
```

```
[1, 5, 7, 85, 23]
[6, 56, 92]
```

Lambdas

Los `lambdas` son funciones anónimas (es decir, funciones que no necesitan de un nombre como las que hemos visto antes) para operaciones básicas de una sola linea. Las veremos mucho en los temas siguientes juntamente con la librería `scipy` .

In [122]:
```
fl1 = lambda x: x + 2
fl1(5)
```

Out[122]:
```
7
```

In [123]:
```
fl2 = lambda x, y: x**2 + y**3
fl2(2, 5)
```

Out[123]:
```
129
```

In [124]:
```
fl3 = lambda k: f'Esto es un ejemplo para {k.capitalize()}!'
fl3('toNI')
```

Out[124]:
```
'Esto es un ejemplo para Toni!'
```

Clases

Las clases son estructuras que pueden contener variables y métodos.

La sintaxis de una clase es:

```
class Nombre():
  def __init__(self, x):
    ...
  def metodo(self, y):
    ...
```

Y se puede instanciar de la siguiente forma:

```
a = Nombre('valor para x')
```

A tener en cuenta:

- En `class Nombre():` se pueden referenciar clases "padre" dentro del paréntesis (la clase `Nombre` hereda las propiedades de las clases "padre");
- En `def __init__(self, x):` se definen las variables de la clase. En este caso se ha definido la variable `x`. Esta variable puede accederse dentro de la clase con `self.x` y desde fuera con `a.x`.
- La palabra clave `self` deberá ir siempre como primer parámetro en las funciones pertenecientes a la clase para que puedan acceder a los métodos y variables de la misma.

In [ ]:
```
class Saludador:
    # Constructor
    def __init__(self, nombre):
        self.nombre = nombre  # Crea una instancia de una variable

    # Método

    def saluda(self, grita=False):
        if grita:
            print("HOLA %s!" % self.nombre.upper())
        else:
            print("Hola %s" % self.nombre)
```

In [ ]:
```
g = Saludador("Paco")  # Construye una instancia de la clase Saludador
g.saluda()  # Llama el método de la instancia; Salida: "Hola Paco"
g.saluda(grita=True)  # Llama el método de la instancia; Salida: "HOLA PAC
O"
```

```
Hola Paco
HOLA PACO!
```

In [125]:
```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart


x = Complex(3.0, -4.5)
x.r, x.i
```

Out[125]:
```
(3.0, -4.5)
```

In [ ]:
```
class Rectangulo:
    def __init__(self, lado1, lado2):
        self.lado1 = lado1
        self.lado2 = lado2

    def area(self):
        return self.lado1 * self.lado2

    def __str__(self):
        return f"El rectángulo con lados {self.lado1} y {self.lado2} tiene
un área de {self.area()}"


rectangulo = Rectangulo(2, 3)
print(rectangulo)
```

```
El rectángulo con lados 2 y 3 tiene un área de 6
```

In [ ]:
```
class Cuadrado(Rectangulo):
    def __init__(self, lado1):
        super().__init__(lado1, lado1)


cuadrado = Cuadrado(3)
print(cuadrado)
print("Área:", cuadrado.area())
```

```
El rectángulo con lados 3 y 3 tiene un área de 9
Área: 9
```

Ejercicios

1. Calcula la siguiente expresión: $\frac{1}{\frac{1}{2}*3^3}$

In [ ]:
```

```

```
0.07407407407407407
```

In [131]:
```
1/1/2*3**3
```

Out[131]:
```
0.07407407407407407
```

1. Dado el diccionario vehiculos compuesto de nombres de vehículos y sus pesos en kilogramos, construye una lista con los nombres, escritos en mayúscula, de aquellos vehículos que tienen un peso inferior a 5000 kilogramos. Usa list comprehension.

In [ ]:
```
vehiculos = {
    "Sedan": 1500,
    "SUV": 2000,
    "Pickup": 2500,
    "Ranchera": 1600,
    "Furgo": 2400,
    "Semi": 13600,
    "Bicicleta": 7,
    "Moto": 110,
}
```

```
['SEDAN', 'SUV', 'PICKUP', 'RANCHERA', 'FURGO', 'BICICLETA', 'MOTO']
```

In [5]:
```
vehiculos = {
    "Sedan": 1500,
    "SUV": 2000,
    "Pickup": 2500,
    "Ranchera": 1600,
    "Furgo": 2400,
    "Semi": 13600,
    "Bicicleta": 7,
    "Moto": 110,
}

[i.upper() for i in vehiculos if vehiculos[i]<5000]
```

Out[5]:
```
['SEDAN', 'SUV', 'PICKUP', 'RANCHERA', 'FURGO', 'BICICLETA', 'MOTO']
```

1. Escribe una función que reciba como argumentos hora y minutos expresados en formato de 24 horas y devuelva una tupla del tipo (hora12, minutos, *) que exprese la hora en el formato 12 horas y un tercer elemento que indique si es tarde (*) o mañana (None). Así, por ejemplo, si los argumentos son hora=20, minutos=16, la tupla resultante sería (8, 16, *). En cambio, si los argumentos son hora=8, minutos=16, la tupla resultante sería (8, 16, None).

In [ ]:
```
print(reloj(20, 36))
print(reloj(7, 25))
print(reloj(23, 59))
```

```
(8, 36, '*')
(7, 25, None)
(11, 59, '*')
```

In [7]:
```
def reloj(hora, minutos):
    if hora > 12:
        return hora-12, minutos, "*"
    else:
        return hora, minutos, None

print(reloj(20, 36))
print(reloj(7, 25))
print(reloj(23, 59))
```

```
(8, 36, '*')
(7, 25, None)
(11, 59, '*')
```

1. Usando math.pi, escribe una función que convierta grados en radianes y viceversa. Los argumentos de la función deben ser (valor,'rad') cuando se quiera convertir grados en radianes, mientras que los argumentos serán (valor,'gra') cuando lo que se quiera es convertir radianes en grados.

In [ ]:
```
print(grad2rad(1, "rad"))
print(grad2rad(1, "gra"))
print(grad2rad(180, "rad"))
```

```
0.017453292519943295
57.29577951308232
3.141592653589793
```

In [11]:
```
def grad2rad(n, tipo):
    if tipo == "rad":
        return n*math.pi/180
    elif tipo == "gra":
        return n/math.pi*180

print(grad2rad(1, "rad"))
print(grad2rad(1, "gra"))
print(grad2rad(180, "rad"))
```

```
0.017453292519943295
57.29577951308232
3.141592653589793
```

1. Escribe una clase que represente un cilindro mediante sus dimensiones (radio y altura) y que tenga como métodos el cálculo del volumen, el cálculo del área y el método que permita imprimir toda la información del cilindro al hacer print.

In [ ]:
```
cyl = Cilindro(2, 4)
print(cyl)
```

```
El cilindro de radio 2 y altura 4 tiene el volumen 50.26548245743669 y la s
uperficie 75.39822368615503
```

In [20]:
```
class Cilindro:
    def __init__(self, radio, altura):
        self.radio = radio
        self.altura = altura

    def volumen(self):
        return math.pi*self.radio**2*self.altura

    def area(self):
        return 2*math.pi*self.radio*self.altura+2*math.pi*self.radio**2

    def __str__(self):
        return f"El cilindro de radio {self.radio} y altura {self.altura} t
iene el volumen {self.volumen()} y la superficie {self.area()}"

cyl = Cilindro(2, 4)
print(cyl)
```

```
El cilindro de radio 2 y altura 4 tiene el volumen 50.26548245743669 y la s
uperficie 75.39822368615503
```