FUNDAMENTOS DE COMPUTACIÓN
Grado en ingeniería en tecnologías Industriales
Universidad de Cantabria

Práctica 8 (laboratorio)
Noviembre 2019

Funciones

- Funciones predefinidas de Python

  Son llamadas funciones las instrucciones mostradas a continuación, que ya hemos utilizado:

```
 >>> type(777777777)
<class 'int'>
>>> print('Hola mundo,', 2019)
Hola mundo, 2019
>>> complex(5, 7.8)
(5+7.8j)
>>> nombre = input()

>>> range(1, 24, 5)
range(1, 24, 5)
```

  > Una función se invoca escribiendo su nombre o identificador seguido de los argumentos o parámetros de la función (que pueden ser varios, uno o ninguno) entre paréntesis y separados por una coma.

- Programar e invocar funciones

  Los ejemplos anteriores son funciones *predefinidas* en el lenguaje Python .

```
1  def positivo(x):
2      '''
3  Entrada: x, un numero real, class float
4  Devuelve el booleanoTrue si es positivo, False en caso contrario
5  '''
6      print('Instrucciones dentro de la funcion positivo')
7      return x > 0
```

  - La primera línea de la definición de una función contiene la palabra reservada def seguida del nombre de la función y entre paréntesis los argumentos (cero o más) y finalmente los dos puntos :
  - Lo que está encerrado entre comillas triples es la **documentación** de la función (opcional), pero recomendable.
  - Las instrucciones que forman la función se escriben con sangría con respecto a la primera línea, denominado el **cuerpo** de la función.
  - Si durante la ejecución de una función se alcanza la instrucción return esta ejecución se termina y el intérprete utiliza el valor indicado.

  Invocando la función:

```
>>> positivo(0)
Instrucciones dentro de la funcion positivo
False
>> positivo(3.14)
Instrucciones dentro de la funcion positivo
True
>>> positivo('pi')
Instrucciones dentro de la funcion positivo
Traceback (most recent call last):
.....
TypeError: '>' not supported between instances of 'str' and 'int
```

La siguiente función tiene dos argumentos:

```
def rep_base(n, b):
    '''Devuelve la cadena con la representacion
     de el numero n en la base b '''
    l =''
    while n != 0:
        l = str(n%b)+l
        n = n // b
    return l
```

Invocando la función:

```
>>> rep_base(12,3)
'110'
>>> rep_base(17,3)
'122'
>>> rep_base(17,2)
'10001'
>>> bin(17)[2:] == rep_base(17,2)
True
```

- Variables locales y globales
  - Variables **locales**, sólo existen en la propia función, incluso cuando en el programa exista una variable con el mismo nombre. Por ejemplo, las variable fact y i introducidas en el cuerpo de la función de arriba.
  - Variables **globales**, si a la variable se le ha asignado valor en el programa principal. Por ejemplo, las variables fact, i y n en el programa de abajo.

```
1 fact = 6; i = 'Peña la Milana'; n = 20
2 def factorial(n):
3     ''' Programa para calcular el factorial de un
4       numero entero positivo o cero '''
5     fact = 1
6     for i in range(1, n+1):
7         fact *= i
8     return fact
```

Ejecutamos el programa:

```
>>> f5 = factorial(5)
>>> f5 + factorial(1)
121
>>> i
'Peña la Milana'
>>> fact
6
>>> n
20
```

- NoneType

  En el siguiente programa, no aparece en el cuerpo de la función la sentencia return, entonces devuelve el valor NoneType

---

```
1 def positivo1(x):
2     '''
3 Entrada: x, un numero real, class float
4 Devuelve None
5 '''
6     print("El programa ha llegado, hasta la linea 10")
7     x > 0
```

Ejecutando:

```
>>> f = positivo1(3)
El programa ha llegado, hasta la linea 10
>>> type(f)
<class 'NoneType'>
>>> f
>>>
```

- RETURN VS PRINT
  - return solo tiene significado dentro de la función, solo se ejecuta una vez y tiene un valor asociado.
  - La función print puede usarse fuera del cuerpo de la función, ejecutarse varias veces dentro del cuerpo de la función.

```
1  def representacion_numero_en_base():
2      ''' Devuelve la lista con  la representacion de un numero n positivo
3         en una base entera b, ambas leidas por teclado '''
4      print('Dame el numero: ', end = ' ')
5      n = int(input())
6      print('Dame la base: ',  end = ' ')
7      b = int(input())
8      L = []
9      if b < 1:
10         return L
11     while b <= n:
12         L =  [n % b] + L
13         n = n // b
14     return [n]+ L
```

Si lo ejecutamos:

```
>>> representacion_numero_en_base()
Dame el numero:  12
Dame la base:  2
[1, 1, 0, 0]
>>> a = representacion_numero_en_base()
Dame el numero:  2019
Dame la base:  1024
>>> a
[1, 995]
>>> representacion_numero_en_base()
Dame el numero:  2019
Dame la base:  -2
[]
>>> type(representacion_numero_en_base)
<class 'function'>
```

- ARGUMENTOS DE UNA FUNCIÓN
  - Los argumentos de la función pueden estar fijos. En el siguiente ejemplo el segundo argumento por defecto es la letra a, pero podemos invocar la función con otro argumento.

```
1  def letras(palabra, letra='a'):
2      ''' Entrada una cadena y una letra (por defecto la letra a)
3  Devuelve el numero de veces que la cadena contiene la letra '''
4      i = 0
5      for x in palabra:
```

```
        if x == letra:
            i += 1
    return i
```

Invocando la función:

```
>>> letras('abracadabra')
5
>>> letras('abracadabra', 'b')
2
>>> letras('abracadabra',1)
0
```

- Los argumentos pueden ser objetos de cualquier clase, incluso de la clase FUNCTION, funciones.

```
def func_a():
    print('funcion_a')
    return 8
def func_b(y):
    print('funcion_b')
    return y
def func_c(z):
    print('funcion_c')
    return z()
```

Invocando las funciones:

```
>>> print(5+func_b(2))
funcion_b
7
>>> print(func_c(func_a))
funcion_c
funcion_a
8
```

- MODULARIDAD

  Un principio útil para diseñar un programa consiste en dividir la tarea que se quiere alcanzar en procesos más sencillos e independientes que, combinados, permiten resolver el problema: **modularidad**.

  Es interesante poder definir nuevas funciones para construir a partir de ellas un programa complejo.

$$coseno(x) = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$

```
def coseno(x, m = 100):
    cos = 0
    for j in range(m+1):
        cos += (-1)**j * x**(2 * j) / factorial(2 * j)
    return cos
```

Ejercicios

1. Es sabido que

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = \sum_{n=0}^{\infty} \frac{x^n}{\prod_{j=1}^{n} j}.$$

   Escribe la función `exponencial` para aproximar esta serie hasta un término dado, que tenga por argumentos el número x y el termino de sumandos (que por defecto sea 25). Por ejemplo:

```
>>> exponencial(1)
2.7182818284590455
>>> exponencial(2.3,100)
9.974182454814718
>>> exponencial(345,0)
1
```

2. Implementar una función de acuerdo con la documentación descrita abajo.

```
def zigzag(s1,s2):
""" Entrada: dos cadenas s1 y s2 ---> SALIDA:   una cadena
  Requisito previo: las dos cadenas tienen la misma longitud len(s1)==len(2).
   Devuelve una cadena conteniendo los caracteres alternativos de s1 y s2
   comenzando con s1[0], entonces s2[1], s1[2], s2[3],...
"""
```

Por ejemplo:

```
>>> zigzag('abc', '123')
'a2c'
>>> zigzag('abcd', '1234')
'a2c4'
```

3. Con el módulo DATETIME de PYTHON podemos conocer la fecha y la hora actual:

```
>>> import datetime
>>> datetime.datetime.now()
datetime.datetime(2019, 11, 13, 7, 5, 28, 827203)
>>> str(datetime.datetime.now())
'2019-11-13 07:06:28.491984'
>>> b = datetime.date.today()
>>> b
datetime.date(2019, 11, 13)
>>> str(b)
'2019-11-13'
```

Escribe una función denominada `fecha_actual` que no tenga ningún argumento y devuelva la cadena conteniendo la fecha actual como se ilustra en el siguiente ejemplo:

```
>>> fecha_actual()
'13 de noviembre de 2019.'
```

4. Una permutación es simplemente un nombre para reordenar. Así, las permutaciones de la cadena 'abc' son 'abc', 'acb', 'bac', 'bca', 'cab', y 'cba'. Observar que la propia cadena es una permutación de ella misma (la permutación trivial). En este ejercicio se pide escribir una función `testpermutacion` que tenga como entrada dos cadenas `s1` y `s2` y devuelva el booleano `True` si la cadena `s2` es una permutación de la cadena `s1` o el booleano `False` en caso contrario. Como requisito todos los caracteres de la cadena `s1` deben ser distintos.

```
>>> testpermutacion("acb","bca")
True
>>> testpermutacion("abc","baa")
False
>>> testpermutacion("abc","ba2")
False
```

5. Define una función llamada `suma_segun` que tome un solo argumento y devuelva:

   - Si el argumento es una lista, la suma de los números enteros impares que contenga.
   - Si es una «tupla», la suma de los números enteros pares que contenga.
   - Cero, en otro caso.

   De este modo, la función debe comportarse así:

```
>>> suma_segun([1,2,3])==4
True
>>> suma_segun((1,2,3))-2
0
>>> suma_segun('123')>1
False
>>> suma_segun(7)+suma_segun([7])
7
>>> suma_segun([True,5.3,'rst',[1,3,5]])
0
```

6. Diseña una función `numeros_pitagoricos` que tenga como argumento dos enteros positivos y devuelva el booleano `True` o `False` dependiendo si la suma de sus cuadrados es un cuadrado o no lo es; por ejemplo $3^2 + 4^2 = 5^2$.

```
>>> numeros_pitagoricos(3,4)
True
>>> numeros_pitagoricos(3,1)
False
```

7. Escribe una función `tiempo` que tenga como argumento un entero positivo n y devuelva la lista formada por *años, días, horas, minutos* y *segundos* que contiene el entero n.

```
>>> tiempo(65)
[0,0,0,1,5]
>>> tiempo(123456789)
[3,333,21,33,9]
```

8. Este ejercicio está dedicado a la construcción de un programa para hacer algunos cálculos sobre triángulos en el plano, que vendrán definidos por las coordenadas de sus vértices. Utilizaremos entonces el tipo(class) `tuple`, y más concretamente, *tuplas* formadas por dos números.

   Entonces el programa solicitará introducir por teclado las coordenadas y tendremos que implementar una función que tenga como argumento una cadena de la forma "$numero1, numero2$" y como salida la tupla $(numero1, numero2)$.

```
def evalua(cadena):
    """ Evalua una cadena """
  return float(cadena.split(",")[0]),float(cadena.split(",")[1])
```

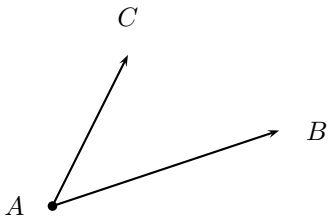
   1. En primer lugar, definimos una función para calcular la distancia entre dos puntos. Nos será útil para conocer la longitud de los lados de los triángulos que manejemos.

```
def distancia(v,w):
    """Distancia entre dos vectores bidimensionales."""
    return ((v[0]-w[0])**2+(v[1]-w[1])**2)**.5
```

      Comprueba el funcionamiento de esta función y utilízala en algún programa que, por ejemplo, pida al usuario tres puntos (A, B y C) y diga cuál de los dos primeros está más cerca de C.

   2. ¿Cómo saber si tres puntos A, B y C definen un triángulo? Cada una de las siguientes condiciones es necesaria y suficiente:

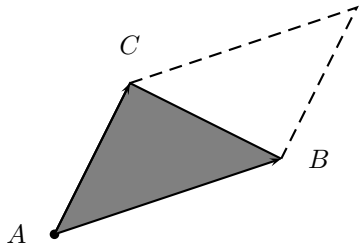
      - no hay ninguna recta que pase por los tres
      - AB y AC son linealmente independientes
      - rango $[AB|AC] = 2$
      - $\det [AB|AC] \neq 0$

      En las dos últimas, $[AB|AC]$ denota la matriz 2×2 cuyas columnas son los vectores AB y AC.
      Utiliza la función siguiente para hacer un programa que decida si tres puntos del plano definen o no un triángulo:

```
def deter(v1,v2,v3):
    return (v2[0]-v1[0])*(v3[1]-v1[1])-(v2[1]-v1[1])*(v3[0]-v1[0])
```

¿Podemos emplear la función `deter` para calcular el área de un triángulo?



3. Veamos un ejemplo de la ejecución del programa que queremos construir:

```
$ python triangulo.py
--------------------------------
       BIENVENIDO
--------------------------------

Describa un triángulo del plano:
Primer vértice  (x,y): 0,0
Segundo vértice (x,y): 2,1
Tercer vértice  (x,y): -3,-1.5
CUIDADO: Esos puntos no definen un triángulo.

Describa un triángulo del plano:
Primer vértice  (x,y): 0,0
Segundo vértice (x,y): 2,1
Tercer vértice  (x,y): 1,2

Escoja una opción:
1.| Longitud de los lados
2.| Perímetro
3.| Área
4.| Ángulos
5.| Situar un cuarto punto en relación al triángulo
6.| Cambiar el triángulo
7.| Abandonar el programa
3
Área: 1.5

Escoja una opción:
1.| Longitud de los lados
2.| Perímetro
3.| Área
4.| Ángulos
5.| Situar un cuarto punto en relación al triángulo
6.| Cambiar el triángulo
7.| Abandonar el programa
5
Escriba las coordenadas del cuarto punto: 1,1
Dentro
```

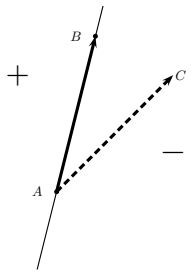
Un modo de organizar el programa:

```
_____ triangulo.py _____
acabar=False
while not(acabar):

    # Carga del triángulo
    while True:
        v1=evalua(input());v2=evalua(input
        v3=evalua(input())
        if define(v1,v2,v3): break
        print('¡OJO!')

    # Cálculo de los lados
    l1=distancia(v2,v3); (···)

    # Menú
    while True:
        opcion=lanza_menu()
        if opcion==1:
            (···)
        elif (···)
        elif opcion==6: break
        elif opcion==7:
            acabar=True
            break
_____________________________
```

1) ¿Y cómo saber si un punto Z queda dentro (consideremos incluido el borde) o fuera del triángulo? Estará dentro si (y solo si) se cumplen las tres condiciones siguientes:
   - La recta definida por A y B divide al plano en dos semiplanos. El vértice restante C y Z están en el mismo semiplano (o Z está precisamente en la recta).
   - La recta definida por A y C divide al plano en dos semiplanos. El vértice restante B y Z están en el mismo semiplano (o Z está precisamente en la recta).

- La recta definida por B y C divide al plano en dos semiplanos. El vértice restante A y Z están en el mismo semiplano (o Z está precisamente en la recta).

¿Pero cómo determinamos si se cumple una de estas condiciones? Mediante dos determinantes: `deter(A,B,Z)` será 0 si Z está en la recta y tendrá el mismo o distinto signo que `deter(A,B,C)` según Z y C estén en el mismo o distintos semiplanos.



9. Las piscinas de Fontecha utilizan la tabla de tarifas siguiente:

	ENTRADA		BONO	
EDAD	— 10	11+	— 10	11 +
vecino	0	1'5	0	15
hijo del pueblo	0	2	0	26
forastero	0	2'5	0	47

Escribe un programa para la taquilla:

```
$ python taquilla.py
-------------------------------------------
----------- PISCINAS MUNICIPALES -----------
-------------------------------------------

Edad: 6
Tarifa gratuita
¿Entrada(E) o bono(B)? B
Nombre: Felipe Mediavilla
Imprimiendo tarjeta...
*******************************************
            PISCINAS MUNICIPALES
*******************************************
 BONO                                  2019
             Felipe Mediavilla

 Hasta 10 años                Tarifa gratuita
*******************************************
```

```
$ python taquilla.py
-------------------------------------------
----------- PISCINAS MUNICIPALES -----------
-------------------------------------------
Edad: 57
¿Entrada(E) o bono(B)? E
¿Vecino(V), h. pueblo(H) o forastero(F)? H
Son 2, a pagar en efectivo.
Importe recibido: 10
8 de vuelta
Imprimiendo tarjeta...
*******************************************
            PISCINAS MUNICIPALES
*******************************************
 ENTRADA            14 de noviembre del 2019


 Adulto                            Son 2 euros
*******************************************
```

Ampliar el programa para continuar despachando billetes hasta que el usuario indique lo contrario, momento en el que el programa podría imprimir el importe total recaudado. Utilizar la función del ejercicio 3 anterior para mostrar la fecha de la tarjeta.

10. Se tiene un número N de bombillas y un número N de interruptores situados en un panel, que cambian el estado de las bombillas de encendido a apagado o viceversa según el estado de la bombilla y siguiendo este esquema:

- El primer interruptor cambia el estado de las bombillas 1,2,3,..,i,.. hasta la bombilla N.
- El segundo interruptor cambia el estado las bombillas 2,4,6,...,2i,.. hasta la bombilla N.
- El tercer interruptor cambia el estado de la bombilla 3,6,9, ...,3i,.. hasta la bombilla N.

  ...
- El interruptor N sólo cambia el estado de la bombilla N.

Por ejemplo: si N=4 y representado el estado de las bombillas por [0,0,1,0], siendo 0 el estado apagado y 1 encendido, pulsando los interruptores 1,2,3,4 se pasaría por los siguientes estados:

$$[1,1,0,1] \rightarrow [1,0,0,0] \rightarrow [1,0,1,0] \rightarrow [1,0,1,1]$$

Quedando la bombillas 1,3 y 4 en estado encendido.

Y si N=5, y el estado de las bombillas es [1,0,1,0,1], pulsando los interruptores 1,2,3,4 y 5 se pasaría por los siguientes estados:

$$[0,1,0,1,0] \to [0,0,0,0,0] \to [0,0,1,0,0] \to [0,0,1,1,0] \to [0,0,1,1,1]$$

Quedando la bombillas 3 ,4 y 5 en estado encendido.

Escribir una función que dado una lista con el estado asociado a cada bombilla, devuelva la lista del estado final de las bombillas si pulsamos ordenadamente todos interruptores desde 1 hasta la longitud de la lista.