Funciones definidas por el usuario

Como en todo código de programación, se puede presentar el caso en el que un conjunto de instrucciones se ejecutan en repetidas oportunidades. Esto se puede establecer en un lazo de repetición, pero puede que inclusive sea el lazo el que se repita en varias secciones del código.

Por otro lado, a veces para resolver un problema de programación, es mejor utilizar la técnica de "divide y venceras", de tal forma que un problema complejo se separe en sub-problemas de fácil solución de forma independiente.

Es en esto casos que la idea de las funciones resulta útil: secciones separadas de código que permiten encapsular una rutina de ejecución.

```
def nombre_funcion(argumentos):
    <Instrucciones>
    return <valores>
```

Por ejemplo, si se quiere construír una función que retorne grados Fahrenheit a partir de valores en grados centigrados:

In [1]:

```
def F(C):
    return (9.0/5) * C + 32
```

Para probar la función, será necesario llamar a la función desde un script, que se puede considerar como una función principal (main):

In [4]:

```
# Se crea una lista de grados centigrados entre 0 y 100 en pasos de 5
gradosC = list(range(0,101,5))

# Se halla una lista con los grados Fahrenheit
gradosF = []
for grad in gradosC:
    gradosF.append(F(grad))

# Se imprimen los resultados
# Recuerde: zip() es una funcion que desempaqueta dos o mas iterables
for gC, gF in zip(gradosC, gradosF):
    print("{:3}°C -> {:4.1f}F".format(gC, gF))
```

```
  0°C -> 32.0F
  5°C -> 41.0F
 10°C -> 50.0F
 15°C -> 59.0F
 20°C -> 68.0F
 25°C -> 77.0F
 30°C -> 86.0F
 35°C -> 95.0F
 40°C -> 104.0F
 45°C -> 113.0F
 50°C -> 122.0F
 55°C -> 131.0F
 60°C -> 140.0F
 65°C -> 149.0F
 70°C -> 158.0F
 75°C -> 167.0F
 80°C -> 176.0F
 85°C -> 185.0F
 90°C -> 194.0F
 95°C -> 203.0F
100°C -> 212.0F
```

Una funcion puede manejar multiples argumentos:

In [6]:
```
def yfunc(t, v0):
    g = 9.81
    return v0*t - 0.5*g*t**2
```

Note que *g* es una variable local con un valor fijo, mientras que *t y* v0* son argumentos y por lo tanto tambien variables locales.

In [9]:
```
print(yfunc(0.1, 6))
print(yfunc(0.1, v0 = 6))
print(yfunc(t = 0.1, v0 = 0.6))
print(yfunc(v0 = 0.6, t = 0.1))
```

```
0.55095
0.55095
0.010949999999999988
0.010949999999999988
```

Note que en el ejemplo anterior, los argumentos se pueden especificar de dos formas: directamente separando los valores por comas (notación posicional) de forma que al llamar a la función, esta sabrá que valor le corresponde a que argumento, o indicando que argumento específico tiene que valor (notación por palabra clave o *keyword*). Esta ultima forma de llamar tiene la ventaja de que es una llamada más clara, ademas de no requiere colocar los arguementos en orden.

Así también, una función puede retrnar más de un valor. Por ejemplo, estamos interesados en evaluar tanto y(t) del caso anterior, asi como su derivada:
$$ \frac{dy}{dt} = v_{0} - gt $$

In [10]:
```
def yfunc(t, v0):
    g = 9.81
    y = v0*t - 0.5*g*t**2
    dydt = v0 - g*t
    return y, dydt
```

Para llamar a esta función, requeriremos especificarle las dos variables de salida:

In [18]:
```
posicion, velocidad = yfunc(t = 1.5, v0 = 10)
print("Posicion = ", posicion, "metros")
print("Velocidad = ", velocidad, "metros por segundo")
```

```
Posicion =  3.963749999999999 metros
Velocidad =  -4.715 metros por segundo
```

Entonces ahora podemos utilizar esta función para crear una tabla:

In [21]:

```
# Creamos una lista por comprension de tiempos
t_list = [0.05 * i for i in range(10)]
for t in t_list:
    pos, vel = yfunc(t, v0 = 5)
    print("t = {:.2f}\tposicion = {:4f}\tvelocidad = {:.4f}".format(t, pos, vel))
```

```
t = 0.00	posicion = 0.000000	velocidad = 5.0000
t = 0.05	posicion = 0.237737	velocidad = 4.5095
t = 0.10	posicion = 0.450950	velocidad = 4.0190
t = 0.15	posicion = 0.639638	velocidad = 3.5285
t = 0.20	posicion = 0.803800	velocidad = 3.0380
t = 0.25	posicion = 0.943437	velocidad = 2.5475
t = 0.30	posicion = 1.058550	velocidad = 2.0570
t = 0.35	posicion = 1.149138	velocidad = 1.5665
t = 0.40	posicion = 1.215200	velocidad = 1.0760
t = 0.45	posicion = 1.256737	velocidad = 0.5855
```

Para definir de forma formar una función, es necesario que esta incluya un "docstring"; esto es, el texto de ayuda de la funcion

In [26]:

```
def yfunc(t, v0):
    '''
    Obtiene la posicion y la velocidad de una particula
    en un instante de tiempo y con una velocidad inicial

    t = tiempo de evaluacion
    v0 = velocidad inicial
    return: posicion, velocidad
    '''
    g = 9.81
    y = v0*t - 0.5*g*t**2
    dydt = v0 - g*t
    return y, dydt
```

In [27]:

```
help(yfunc)
```

```
Help on function yfunc in module __main__:

yfunc(t, v0)
    Obtiene la posicion y la velocidad de una particula
    en un instante de tiempo y con una velocidad inicial

    t = tiempo de evaluacion
    v0 = velocidad inicial
    return: posicion, velocidad
```

Funciones lambda

Existe una forma rápida de construír una función que a veces resulta conveniente:

In [37]:

```
f = lambda x: x**2 + 4
print("2**2 + 4 =",f(2))
```

```
2**2 + 4 = 8
```

Esta es una función "anónima" y que ha sido asignada a una variable. Una función sin nombre se conoce con el nombre de "funcion lambda". Esto es equivalente a escribir:

```
def f(x):
    return x**2 + 4
```

En general:

```
def g(arg1, arg2, arg3, ...):
    return expresion
```

Se puede escribir como:

```
g = lambda arg1, arg2, arg3,...: expresion
```

Retomemos el caso de la función de conversión de grados utilizando la función *map*. Esta función permite modificar todos los elementos de una lista por una función de la forma:

```
map(funcion, lista)
```

En este caso, se utilizara la función lambda:

```
 lambda C: (9.0/5) * C + 32
```

Par que afecte a cada valor de la lista gradosC:

In [33]:

```
# Se crea una lista de grados centigrados entre 0 y 100 en pasos de 5
gradosC = list(range(0,101,5))

# Se crea una lista con sus equivalente de grados Fahrenheit sin recurrir a un lazo
gradosF = list(map(lambda C: (9.0/5) * C + 32, gradosC))

# Se imprimen los resultados
for gC, gF in zip(gradosC, gradosF):
    print("{:3}°C -> {:4.1f}F".format(gC, gF))
```

```
  0°C -> 32.0F
  5°C -> 41.0F
 10°C -> 50.0F
 15°C -> 59.0F
 20°C -> 68.0F
 25°C -> 77.0F
 30°C -> 86.0F
 35°C -> 95.0F
 40°C -> 104.0F
 45°C -> 113.0F
 50°C -> 122.0F
 55°C -> 131.0F
 60°C -> 140.0F
 65°C -> 149.0F
 70°C -> 158.0F
 75°C -> 167.0F
 80°C -> 176.0F
 85°C -> 185.0F
 90°C -> 194.0F
 95°C -> 203.0F
100°C -> 212.0F
```

Asi también, se puede utilizar la función *filter* en combinacion con una función lambda para seleccionar los elementos de una lista que cumplan con cierto criterio de la forma:

```
filter(funcion, lista)
```

En esta caso queremos obtener las valores de temperatura Fahrenheit que sean pares:

```
lambda F: F % 2 == 0
```

In [40]:

```
# Se filtran valores de una lista sin recurrir a un lazo ni a una instruccion condicional
F_pares = list(filter(lambda F: F % 2 == 0, gradosF))

for gF in F_pares:
    print("{:4.1f}F".format(gF))
```

```
32.0F
50.0F
68.0F
86.0F
104.0F
122.0F
140.0F
158.0F
176.0F
194.0F
212.0F
```

Las funciones *lambda*, combinación con las funciones *map* y *filter*, eliminan lazos e instrucciones condicionales, haciendo que el código sea mas *"Pythonico"*

Funciones recursivas

Muchas operaciones se pueden expresar de forma recursiva: esto es, utilizando la propia definicion como parte de la definición de una operacion. Por ejemplo, la multiplición de dos valores positivos tienen las siguientes reglas:

```
(1) : n * 0 = 0
(2) : n * m = n + (n * m-1)
```

Esto quiere decir que 2 * 4 se puede expresar de la forma:

```
2 * 4 = 2 + (2 * 3) = 2 + 2 + (2 * 2) = 2 + 2 + 2 + (2 * 1) = 2 + 2 + 2 + (2 * 0)
```

La ultima operacion (2 *0) retorna 0 por ser el caso (1): n* 0 = 0, por lo tanto, regresando, se tiene:

```
2 * 4 = 2 + 2 + 2 + 2 + 0
```

Esta definición recursiva se puede utilizar para construír una función que se llame a si misma y que "regrese sobre sus pasos" utilizando el caso (1) como punto de retorno:

In [45]:

```
def multiplica(a, b):
    if b == 0:
        return 0
    else:
        return a + multiplica(a, b-1)
```

In [50]:

```
for num in range(1,11):
    print("9 x {} = {}".format(num, multiplica(9,num)))
```

```
9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
9 x 10 = 90
```

La función multiplica realiza el producto haciendo sumas sucesivas. ¿Puede construír una función recursiva que realice la division de dos números utilizando una defincón recursiva del cociente a traves de restas sucesivas?

In [8]:

```
#---------------------- MODULOS EN PYTHON ---------------------

#UTILICE SPYDER

# Se puede crear un archivo de modulos definido por el usuario.
# Un modulo es un archivo que contiene instrucciones y definiciones
# Vamos crear un modulo de funciones utiles para las tareas relacionadas
# al calculo en ingenieria electronica

#docstring del módulo:

'''Modulo de funciones electronicas

Funciones en modulo:
 - r_serie(r1, r1, ..., rn)
 - r_paralelo(r1, r2, ..., rn)
 '''

#se crean las funciones del módulo:
def r_serie(*resistencias):
    '''Funcion que obtiene la resistencia equivalente de valores resistivos
    en serie'''
    suma = 0
    for R in resistencias:
        suma += R

    return suma

def r_paralelo(*resistencias):
    '''Funcion que obtiene la resistencia equivalente de valores resistivos
    en serie'''
    suma = 0
    for R in resistencias:
        suma += 1/R

    return 1/suma

# A este archivo lo guardamos con el nombre electronica.py (verificar que el
# directorio de trabajo sea el mismo donde se ha guaradado el archivo de modulo)
# Ahora podemos probar el modulo en la consola

#comprobar en sypder:
#import electronica
#print(electronica.r_serie(10, 10))

#print(electronica.r_paralelo(10,10))
```

In [10]:

```
# Si se hacen cambios al archivo de modulo y se quieren probar las modificaciones
# es necesario refrescar la importacion del modulo

#comprobar en sypder:

#import importlib
#importlib.reload(electronica)
```

In [ ]:

```
# Los modulos definidos por el usuario se importan de la misma forma que los modulos
# del sistema:

#import electronica
#import electronica as e
#from electronica import r_serie
#from electronica import *
```

In [12]:

```
# Y la funcion dir permite explorar las funciones del modulo

#probar en spyder:
#dir(electronica)
```

In [ ]:

```
# Pruebe el atributo __doc__ del modulo que contiene el docsting del modulo

#probar en spyder:
#electronica.__doc__
```

In [ ]:

```
# Puede revisar como utiliza los docstrings la funcion help para dar un formato
# de documentacion de ayuda

#probar en spyder:
#help(electronica)
```

In [13]:

```
#-------------------------- El Zen de Python -------------------------------
# De todo lo visto hasta ahora, debe quedar claro que detras del diseño de lenguaje de
# programacion Python hay una filosofia de hacer las cosas. Esta esta resumida en un
# documento llamado El Zen de Python.
#
# Es algo asi como los mandamientos de Pyton y se puede acceder a ellos con la instruccion:
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

In [ ]: