El lenguaje de programación *Python*

El objetivo de esta práctica es realizar una serie de ejercicios de familiarización con el lenguaje de programación *Python*.

Para evaluar las celdas se puede usar uno de los siguientes tres comandos:

- Control-Enter: evalúa la celda y permanece en ella
- Mayúsculas-Enter: evalúa la celda y selecciona la siguiente celda
- Alt-Enter: evalúa la celda e inserta una celda vacía justo debajo

Ejercicio 1

Se pide definir una función `cuadrados` que, dada una secuencia `sec` de números, devuelva la lista de los cuadrados de esos números, en el mismo orden.

Ejemplos:

- `cuadrados([2, -1.2, 3e2, 1j])` debe devolver `[4, 1.44, 90000.0, (-1+0j)]`
- `cuadrados(i for i in range(10))` debe devolver `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`

In [1]:
```
def cuadrados(sec):
    ret = []
    for elem in sec:
        ret.append(elem*elem) # elem^2
    return ret
```

In [2]:
```
cuadrados([2, -1.2, 5, 7])
```

Out[2]:
```
[4, 1.44, 25, 49]
```

In [3]:
```
cuadrados(range(10))
```

Out[3]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Versión reducida usando listas por comprensión.

In [4]:
```
def cuadrados_red(sec):
    return [elem*elem for elem in sec]
```

In [5]:
```
cuadrados_red([2, -1.2, 5, 7])
```

Out[5]:
```
[4, 1.44, 25, 49]
```

In [6]:
```
cuadrados_red(range(10))
```

Out[6]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Ejercicio 1.1

Se pide definir una función `son_primos` que dada una secuencia sec de numeros, devuelva una lista de tuplas que con tienen el numero original acompañado de un valor lógico

`True/False` indicando si el número es primo o no.

Ejemplo, dado una lista `[2,3,4,5,6,10,13]` devuelve `[(2, True), (3, True), (4, False), (5, True), (6, False), (10, False), (13, True)]`

Podemos empezar diseñando una funcion `es_primo` que dado un valor numérico me diga si es primo o no.

Notas

- Descomponer el problema en más de una función puede facilitar la tarea
- Operador división entera `//`
- Operador resto `%`
- Creación de tuplas `variable = (valor1, valor2, valor3)`

Descomposición del problema:

1. Función `es_primo`
2. Función `son_primos`

In [87]:
```
def es_primo(n):
    if n < 2:
        return False
    #for num in range(2,n):
    for i in range(2,(n // 2)+1):
        if n % i == 0:
            return False
    return True
```

In [8]:
```
def son_primos(sec):
    ret = []
    for elem in sec:
        ret.append((elem, es_primo(elem)))
    return ret
```

In [9]:
```
son_primos([0,1,2,3,4,5,6,10,13])
```

Out[9]:
```
[(0, False),
 (1, False),
 (2, True),
 (3, True),
 (4, False),
 (5, True),
 (6, False),
 (10, False),
 (13, True)]
```

Versión reducida de la función `son_primos` usando listas por comprensión.

In [84]:
```
def son_primos(sec):
    return [(elem, es_primo(elem)) for elem in sec]
```

In [11]:
```
son_primos([0,1,2,3,4,5,6,10,13])
```

Out[11]:
```
[(0, False),
 (1, False),
 (2, True),
 (3, True),
 (4, False),
 (5, True),
```

```
(6, False),
(10, False),
(13, True)]
```

Usando una sola función:

In [2]:
```
def son_primos(sec):
    ret = []
    for n in sec:
        if n < 2:
            ret.append((n, False))
            continue
        for i in range(2,(n // 2)+1):
            if n % i == 0:
                ret.append((n, False))
                break
        else:
            ret.append((n, True))
    return ret
```

In [3]:
```
son_primos([0,1,2,3,4,5,6,10,13])
```

Out[3]:
```
[(0, False),
 (1, False),
 (2, True),
 (3, True),
 (4, False),
 (5, True),
 (6, False),
 (10, False),
 (13, True)]
```

¿Y si queremos filtrar los que son primos? Ejemplo, dado una lista `[2,3,4,5,6,10,13]` devuelve `[2,3,5,13]`

In [85]:
```
def filtra_primos(sec):
    return [elem for elem in sec if es_primo(elem)]
```

In [88]:
```
filtra_primos([2,3,4,5,6,10,13])
```

Out[88]:
```
[2, 3, 5, 13]
```

Ejercicio 2

Un número entero positivo se dice que es perfecto si coincide con la suma de todos sus divisores propios (es decir, distintos de él mismo). Se pide definir una función `escribe_perfectos` que, dados dos números enteros positivos $m \leq n$, imprima por pantalla todos los números perfectos entre m y n. Se pide también que se indiquen los divisores de cada número perfecto que se imprima.

Ejemplos:

- `escribe_perfectos(1, 1000)` debe escribir en pantalla:

```
El número 6 es perfecto y sus divisores son [1, 2, 3]
El número 28 es perfecto y sus divisores son [1, 2, 4, 7, 14]
El número 496 es perfecto y sus divisores son [1, 2, 4, 8, 16, 31, 62,
124, 248]
```

Notas

- Así se insertan las variables x e y en una cadena: `print('El valor de x es: {} y el de y es: {}'.format(x, y))`
- Suma de los elementos de una lista `sum(lista)`

Descomposición:

- Repetir por cada número entre m y n:
  - Dado un número, calcular la lista de divisores (exluido dicho numero)
  - sumar divisores y comprobar si es perfecto.
  - Imprimir resultado

Version 1: Descomponiendo el problema en 3 funciones. Tiene el inconveniente de que debemos ejecutar la función `divisores_numero` dos veces. Esta es la menos eficiente, el resto de versiones a continuación son similares.

In [33]:
```
def divisores(n):
    divs = []
    for i in range(1, (n//2)+1):
        if n % i == 0:
            divs.append(i)
    return divs

# Divisores usando listas por comprensión:
# def divisoresRed(n):
#     retrun [i for i in range(1, (n // 2)+1) if n % i == 0]

def es_perfecto(n):
    divs = divisores(n)
    # return sum(divs) == n
    if sum(divs) == n:
        return True
    else:
        return False

def escribe_perfectos(m, n):
    for num in range(m, n+1):
        if es_perfecto(num):
            divs = divisores(num)
            print('El numero {} es perfecto y sus divisores son {}'.format(num, divs
```

In [32]:
```
escribe_perfectos(1, 1000)
```

```
El numero 6 es perfecto y sus divisores son [1, 2, 3]
El numero 28 es perfecto y sus divisores son [1, 2, 4, 7, 14]
El numero 496 es perfecto y sus divisores son [1, 2, 4, 8, 16, 31, 62, 124, 248]
```

Versión 2: función `divisores` y función `escribe_perfectos` .

In [12]:
```
def divisores(n):
    divs = []
    for i in range(1, (n//2)+1):
        if n % i == 0:
            divs.append(i)
    return divs
```

In [13]:
```
def escribe_perfectos(m, n):
    for num in range(m, n+1):
        divs = divisores(num)
```

```
        if sum(divs) == num:
            print('El número {} es perfecto y sus divisores son {}'.format(num, divs
```

In [14]:
```
escribe_perfectos(1, 1000)
```

```
El número 6 es perfecto y sus divisores son [1, 2, 3]
El número 28 es perfecto y sus divisores son [1, 2, 4, 7, 14]
El número 496 es perfecto y sus divisores son [1, 2, 4, 8, 16, 31, 62, 124, 248]
```

Versión 3: función `divisores_perfecto` (devuelve la lista de divisores si el número es perfecto, en caso contrario, devuelve `None`) y función `escribe_perfectos`.

In [15]:
```
def es_perfecto(n):
    divs = []
    for i in range(1, (n // 2)+1):
        if n % i == 0:
            divs.append(i)
    if sum(divs)==n:
        return divs
    else:
        return None
```

In [16]:
```
def escribe_perfectos(m, n):
    for num in range(m, n+1):
        divs = es_perfecto(num)
        # if divs:
        if divs is not None:
            print('El número {} es perfecto y sus divisores son {}'.format(num, divs
```

In [17]:
```
escribe_perfectos(1, 1000)
```

```
El número 6 es perfecto y sus divisores son [1, 2, 3]
El número 28 es perfecto y sus divisores son [1, 2, 4, 7, 14]
El número 496 es perfecto y sus divisores son [1, 2, 4, 8, 16, 31, 62, 124, 248]
```

Versión 4: Todo en una sola función.

In [18]:
```
def escribe_perfectos(m, n):
    for num in range(m, n+1):
        divs = []
        for i in range(1, (num//2)+1):
            if num % i == 0:
                divs.append(i)
        if sum(divs) == num:
            print('El numero {} es perfecto y sus divisores son {}'.format(num, divs

# Versión más eficiente usando un acumulador.
# def escribe_perfectos(m, n):
#     for num in range(m, n):
#         acc = 0
#         divs = []
#         for den in range(1, (num // 2) + 1):
#             if (num % den) == 0:
#                 acc += den
#                 divs.append(den)
#         if num == acc:
#             print('El número {} es perfecto y sus divisores son {}'.format(num, di
```

In [19]:
```
escribe_perfectos(1, 1000)
```

```
El numero 6 es perfecto y sus divisores son [1, 2, 3]
El numero 28 es perfecto y sus divisores son [1, 2, 4, 7, 14]
El numero 496 es perfecto y sus divisores son [1, 2, 4, 8, 16, 31, 62, 124, 248]
```

Ejercicio 3

Consideremos diccionarios cuyas claves son cadenas de caracteres de longitud uno y los valores asociados son números enteros no negativos, como por ejemplo el siguiente diccionario `d` :

In [25]:
```
d = {'z': 7, 'a': 5, 'b': 10, 'c': 12, 'd': 11, 'e': 15, 'f': 20, 'g': 15, 'h': 9, '
```

Se pide definir una función `histograma_horizontal` que, dado un diccionario del tipo anterior, escriba en pantalla el histograma de barras horizontales asociado, imprimiendo las barras de arriba a abajo en el orden que determina la función `sorted` sobre las claves, tal y como se ilustra en el siguiente ejemplo:

- `histograma_horizontal(d)` debe escribir en pantalla:

```
a: *****
b: **********
c: ***********
d: ***********
e: ***************
f: ********************
g: ***************
h: *********
i: *******
j: **
```

Notas:

- Obtener todas las claves de un diccionario: `d.keys()`
- Obtener todos los valores de un diccionario: `d.values()`
- Obtener todos los pares `(clave, valor)` de un diccionario: `d.items()`
- Acceder a un elemento de un diccionario dada su clave: `d[clave]`
- Ordenar una coleccion `sorted(col)`
- Repetir un caracter N veces: `caracter*N`

Versión 1: Usando un bucle para construir la cadena.

In [26]:
```
def histograma_horizontal(d):
    for k in sorted(d.keys()):
        cad = ''
        for i in range(d[k]):
            cad += '*'
        print('{}: {}'.format(k, cad))
```

In [27]:
```
histograma_horizontal(d)
```

```
a: *****
b: **********
c: ***********
d: ***********
e: ***************
f: ********************
g: ***************
h: *********
i: *******
j: **
z: *******
```

Versión 1.1: Usando un bucle `while` para construir la cadena.

In [28]:
```
def histograma_horizontal(d):
    for k in sorted(d.keys()):
        cad = ''
        i = 0
        while i < d[k]:
            cad += '*'
            i += 1
        print('{}: {}'.format(k, cad))
```

In [29]:
```
histograma_horizontal(d)
```

```
a: *****
b: **********
c: ***********
d: ***********
e: ***************
f: ********************
g: ***************
h: *********
i: *******
j: **
z: *******
```

Versión 2: Usando el operador `*` para construir la cadena.

In [30]:
```
def histograma_horizontal(d):
    for k in sorted(d.keys()):
        print('{}: {}'.format(k, d[k]*'*'))
```

In [31]:
```
histograma_horizontal(d)
```

```
a: *****
b: **********
c: ***********
d: ***********
e: ***************
f: ********************
g: ***************
h: *********
i: *******
j: **
z: *******
```

Se pide definir una función `histograma_vertical` que, dado un diccionario del tipo anterior, escriba en pantalla el histograma de barras verticales asociado, imprimiendo las barras de izquierda a derecha en el orden que determina la función `sorted` sobre las claves, tal y como se ilustra en el siguiente ejemplo:

- `histograma_vertical(d)` debe escribir en pantalla:

```
          *
          *
          *
          *
          *
        * * *
        * * *
        * * *
    *   * * *
    * * * * *
  * * * * * *
  * * * * * * *
```

```
      * * * * * * *
      * * * * * * * * *
      * * * * * * * * *
    * * * * * * * * * *
    * * * * * * * * * *
    * * * * * * * * * *
    * * * * * * * * * * *
    * * * * * * * * * * *
    a b c d e f g h i j
```

Versión 1: Usando un bucle para construir la última línea.

In [9]:
```
def histograma_vertical(d):
    max_val = max(d.values())
    for i in range(max_val, 0, -1):
        cad = ''
        for k in sorted(d.keys()):
            if d[k] < i:
                cad += '  '
            else:
                cad += '* '
        print(cad)
    cad = ''
    for k in sorted(d.keys()):
        cad += k+' '
    print(cad)
```

In [10]:
```
histograma_vertical(d)
```

```
          *
          *
          *
          *
          *
        * * *
        * * *
        * * *
    *   * * *
    * * * * *
  * * * * * *
  * * * * * * *
  * * * * * * *
  * * * * * * * *
  * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
a b c d e f g h i j
```

Versión 2: Usando un el método `join` para construir la última línea.

In [11]:
```
def histograma_vertical(d):
    max_val = max(d.values())
    for i in range(max_val, 0, -1):
        cad = ''
        for k in sorted(d.keys()):
            if d[k] < i:
                cad += '  '
            else:
                cad += '* '
        print(cad)
    print(' '.join(d.keys()))
```

In [12]:
```
histograma_vertical(d)
```

```
          *
          *
          *
          *
          *
        * * *
        * * *
        * * *
    *   * * *
    * * * * *
  * * * * * *
  * * * * * * *
  * * * * * * *
  * * * * * * * *
  * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
a b c d e f g h i j
```

Ejercicio 4

La profundidad de una lista anidada es el número máximo de anidamientos en la lista. Se pide definir una función `profundidad` que, dada una lista `l`, devuelva la profundidad de `l`.

Indicación: para saber si un dato es una lista, puede que sea útil la función `isinstance`. En concreto, `isinstance(x, list)` comprueba si `x` es una lista.

Ejemplos:

- `profundidad(3)` debe devolver `0`
- `profundidad([7, 5, 9, 5, 6])` debe devolver `1`
- `profundidad([1, [1, [1, [1, 1], 1], [1, 1]], 1])` debe devolver `4`

In [20]:
```
def profundidad(l, profundidad_actual=0):
    if not l or not isinstance(l, list):
        return profundidad_actual
    return max(profundidad(l[0], profundidad_actual+1), profundidad(l[1:], profundid
```

In [22]:
```
print(profundidad(3))
print(profundidad([7, 5, 9, 5, 6]))
print(profundidad([1, [1, [1, [1, 1], 1], [1, 1]], 1]))
print(profundidad([1, [1, [1, [1, 1], 1], [1, 1]], 1, [1, 1, [1, [1, [[1, [1, 1], 1]
```

```
0
1
4
7
```

Veamos una versión un poco más desarrollada de la misma función.

In [23]:
```
def profundidad(l, profundidad_actual = 0):
    if not l or not isinstance(l, list):
        return profundidad_actual
    else:
        lista_profundidades = []
        for item in l:
            lista_profundidades.append(profundidad(item, profundidad_actual + 1))
    return max(lista_profundidades)
```

In [24]:
```
print(profundidad(3))
print(profundidad([7, 5, 9, 5, 6]))
print(profundidad([1, [1, [1, [1, 1], 1], [1, 1]], 1]))
print(profundidad([1, [1, [1, [1, 1], 1], [1, 1]], 1, [1, 1, [1, [1, [[1, [1, 1], 1]
```

```
0
1
4
7
```

Ejercicio 5

Definir la función compresion(l) que devuelva la lista resultante de comprimir la lista l que recibe como entrada, en el siguiente sentido:

- Si el elemento x aparece n (n > 1) veces de manera consecutiva en l sustituimos esas n ocurrencias por la tupla (n, x)
- Si el elemento x es distinto de sus vecinos, entonces lo dejamos igual

Ejemplos

- `compresión([1, 1, 1, 2, 1, 3, 2, 4, 4, 6, 8, 8, 8])` debe devolver `[[3, 1], 2, 1, 3, 2, [2, 4], 6, [3, 8]]`
- `compresión(["a", "a", "a", "b", "a", "c", "b", "d", "d", "f", "h", "h", "h"])` debe devolver `[[3, 'a'], 'b', 'a', 'c', 'b', [2, 'd'], 'f', [3, 'h']]`

Version 1:

In [80]:
```
def compresion(l):
    ret = []
    repeticiones = 1
    for i in range(len(l)):
        if i+1 < len(l) and l[i] == l[i+1]:
            repeticiones += 1
        elif repeticiones > 1:
            ret.append([repeticiones, l[i]])
            repeticiones = 1
        else:
            ret.append(l[i])
    return ret
```

In [81]:
```
print(compresion([1, 1, 1, 2, 1, 3, 2, 4, 4, 6, 8, 8, 8]))
print(compresion([1, 1, 1, 2, 1, 3, 2, 4, 4, 6, 8, 8, 8, 5]))
print(compresion(["a", "a", "a", "b", "a", "c", "b", "d", "d", "f", "h", "h", "h"]))
print(compresion(["a", "a", "a", "b", "a", "c", "b", "d", "d", "f", "h", "h", "h", "
```

```
[[3, 1], 2, 1, 3, 2, [2, 4], 6, [3, 8]]
[[3, 1], 2, 1, 3, 2, [2, 4], 6, [3, 8], 5]
[[3, 'a'], 'b', 'a', 'c', 'b', [2, 'd'], 'f', [3, 'h']]
[[3, 'a'], 'b', 'a', 'c', 'b', [2, 'd'], 'f', [3, 'h'], 'j']
```

Versión 2:

In [82]:
```
def compresion(l):
    ret = []
    repeticiones = 1
    for i in range(len(l)-1):
        if l[i] == l[i+1]:
            repeticiones += 1
```

```
        elif repeticiones > 1:
            ret.append([repeticiones, l[i]])
            repeticiones = 1
        else:
            ret.append(l[i])
    else:
        if repeticiones > 1:
            ret.append([repeticiones, l[-1]])
        else:
             ret.append(l[-1])
    return ret
```

In [83]:
```
print(compresion([1, 1, 1, 2, 1, 3, 2, 4, 4, 6, 8, 8, 8]))
print(compresion([1, 1, 1, 2, 1, 3, 2, 4, 4, 6, 8, 8, 8, 5]))
print(compresion(["a", "a", "a", "b", "a", "c", "b", "d", "d", "f", "h", "h", "h"]))
print(compresion(["a", "a", "a", "b", "a", "c", "b", "d", "d", "f", "h", "h", "h", "
```

```
[[3, 1], 2, 1, 3, 2, [2, 4], 6, [3, 8]]
[[3, 1], 2, 1, 3, 2, [2, 4], 6, [3, 8], 5]
[[3, 'a'], 'b', 'a', 'c', 'b', [2, 'd'], 'f', [3, 'h']]
[[3, 'a'], 'b', 'a', 'c', 'b', [2, 'd'], 'f', [3, 'h'], 'j']
```

Ejercicio 6

Definir la función descompresión(l) que devuelva la lista l descomprimida, suponiendo que ha sido comprimida con el método del apartado anterior.

Ejemplos

- `descompresión([[3, 1], 2, 1, 3, 2, [2, 4], 6, [3, 8]])` debe devolver `[1, 1, 1, 2, 1, 3, 2, 4, 4, 6, 8, 8, 8]`

Versión 1:

In [36]:
```
def descompresion(l):
    ret = []

    for elem in l:
        if isinstance(elem, list):
            ret.extend(elem[0]*[elem[1]])
        else:
            ret.append(elem)
    return ret
```

In [37]:
```
descompresion([[3, 1], 2, 1, 3, 2, [2, 4], 6, [3, 8]])
```

Out[37]:
```
[1, 1, 1, 2, 1, 3, 2, 4, 4, 6, 8, 8, 8]
```

Versión 2: usando excepciones en vez de if/else.

In [38]:
```
def descompresion(l):
    ret = []

    for elem in l:
        # print(elem)
        try:
            ret.extend(elem[0]*[elem[1]])
        except TypeError:
            ret.append(elem)
    return ret
```

In [39]:
```
descompresion([[3, 1], 2, 1, 3, 2, [2, 4], 6, [3, 8]])
```

Out[39]:
```
[1, 1, 1, 2, 1, 3, 2, 4, 4, 6, 8, 8, 8]
```

Ejercicio 7

Supongamos que queremos simular la trayectoria de un proyectil que se dispara en un punto dado a una determinada altura inicial. El disparo se realiza hacia adelante con una velocidad inicial y con un determinado ángulo. Inicialmente el proyectil avanzará subiendo, pero por la fuerza de la gravedad, en un momento dado empezará a bajar hasta que aterrice. Por simplificar, supondremos que no existe rozamiento ni resistencia del viento.

Se pide definir una clase `Proyectil` que sirva para representar el estado del proyectil en un instante de tiempo dado. Para ello, se necesitan al menos atributos de datos que guarden la siguiente información:

- Distancia recorrida (en horizontal)
- Altura
- Velocidad horizontal
- Velocidad vertical

Además, se pide dotar a la clase `Proyectil` de los siguientes tres métodos:

- obtener_pos_x: devuelve la distancia horizontal recorrida
- obtener_pos_y: devuelve la distancia vertical recorrida
- `actualizar_posicion` : dada una cantidad `t` de segundos, actualiza la posición y la velocidad del proyectil tras haber transcurrido ese tiempo

Una vez definida la clase `Proyectil` , se pide definir una función `aterriza` que, dados los datos de `altura` , `velocidad` , `angulo` e `intervalo` , imprima por pantalla las distintas posiciones por las que pasa un proyectil que se ha disparado con esa `velocidad` , `angulo` (en grados) y una `altura` inicial. Se mostrará la posición del proyectil en cada `intervalo` de tiempo, hasta que aterriza. Además también debe imprimir la altura máxima que ha alcanzado al final de cada intervalo, cuántos intervalos de tiempo ha tardado en aterrizar y el alcance que ha tenido.

Indicaciones:

1. Si el proyectil tiene una velocidad inicial v y se lanza con un ángulo θ, las componentes horizontal y vertical de la velocidad inicial son $v \times \cos(\theta)$ y $v \times \sin(\theta)$, respectivamente.
2. La componente horizontal de la velocidad, en ausencia de rozamiento y viento, podemos suponer que permanece constante.
3. La componente vertical de la velocidad cambia de la siguiente manera tras un intervalo de tiempo t: si vy_0 es la velocidad vertical al inicio del intervalo, entonces al final del intervalo tiene una velocidad $vy_1 = vy_0 - 9.8 \times t$, debido a la gravedad de la Tierra.
4. En ese caso, si el proyectil se encuentra a una altura h_0, tras un intervalo de tiempo t se encontrará a una altura $h_1 = h_0 + vm \times t$, donde vm es la media entre las anteriores vy_0 y vy_1.

Ejemplo:

- `aterriza(30, 1, 20, 0.1)` debe escribir en pantalla:

```
Proyectil en posición(0.0, 30.0)
Proyectil en posición(0.1, 30.0)
Proyectil en posición(0.2, 29.9)
Proyectil en posición(0.3, 29.7)
Proyectil en posición(0.4, 29.4)
Proyectil en posición(0.5, 28.9)
Proyectil en posición(0.6, 28.4)
Proyectil en posición(0.7, 27.8)
Proyectil en posición(0.8, 27.1)
Proyectil en posición(0.8, 26.3)
Proyectil en posición(0.9, 25.4)
Proyectil en posición(1.0, 24.4)
Proyectil en posición(1.1, 23.4)
Proyectil en posición(1.2, 22.2)
Proyectil en posición(1.3, 20.9)
Proyectil en posición(1.4, 19.5)
Proyectil en posición(1.5, 18.0)
Proyectil en posición(1.6, 16.4)
Proyectil en posición(1.7, 14.7)
Proyectil en posición(1.8, 13.0)
Proyectil en posición(1.9, 11.1)
Proyectil en posición(2.0, 9.1)
Proyectil en posición(2.1, 7.0)
Proyectil en posición(2.2, 4.9)
Proyectil en posición(2.3, 2.6)
Proyectil en posición(2.3, 0.2)

Tras 26 intervalos de 0.1 segundos (2.6 segundos) el proyectil ha
aterrizado.
Ha recorrido una distancia de 2.4 metros
Ha alcanzado una altura máxima de 30.0 metros
```

In [25]:
```python
class Proyectil(object):
    def __init__(self, altura, vhorizontal, vvertical):
        self.distancia_recorrida = 0
        self.altura = altura
        self.vhorizontal = vhorizontal
        self.vvertical = vvertical

    def obtener_pos_x(self):
        return self.distancia_recorrida

    def obtener_pos_y(self):
        return self.altura

    def actualizar_posicion(self, t):
        vvertical_final = self.vvertical - 9.8*t
        vmedia = (self.vvertical + vvertical_final)/2
        self.altura += vmedia*t
        self.vvertical = vvertical_final
        self.distancia_recorrida += self.vhorizontal*t
```

In [26]:
```python
from math import cos, sin, radians

def aterriza(altura, velocidad, angulo, intervalo):
    vihorizontal = velocidad*cos(radians(angulo))
    vivertical = velocidad*sin(radians(angulo))
```

```
    print(vihorizontal)
    print(vivertical)
    proyectil = Proyectil(altura, vihorizontal, vivertical)
    altura_max = altura
    t = 0
    while proyectil.obtener_pos_y() > 0:
        t += intervalo
        print('Proyectil en posición({},{})'.format(proyectil.obtener_pos_x(), proye
        proyectil.actualizar_posicion(intervalo)
        if proyectil.obtener_pos_y() > altura_max:
            altura_max = proyectil.obtener_pos_y()
    print('Tras {} intervalos de {} ({} segundos) el proyectil ha aterrizado.'.forma
    print('Ha recorrido una distancia de {}'.format(proyectil.obtener_pos_x()))
    print('Ha alcanzado una altura máxima de {} metros'.format(altura_max))
```

In [27]:
```
aterriza(30, 1, 20, 0.1)
```

```
0.9396926207859084
0.3420201433256687
Proyectil en posición(0,30)
Proyectil en posición(0.09396926207859085,29.985202014332568)
Proyectil en posición(0.1879385241571817,29.872404028665134)
Proyectil en posición(0.28190778623577256,29.6616060429977)
Proyectil en posición(0.3758770483143634,29.352808057330268)
Proyectil en posición(0.46984631039295427,28.946010071662833)
Proyectil en posición(0.5638155724715451,28.4412120859954)
Proyectil en posición(0.657784834550136,27.838414100327967)
Proyectil en posición(0.7517540966287268,27.137616114660535)
Proyectil en posición(0.8457233587073176,26.3388181289931)
Proyectil en posición(0.9396926207859084,25.442020143325667)
Proyectil en posición(1.0336618828644992,24.447222157658235)
Proyectil en posición(1.12763114494309,23.3544241719908)
Proyectil en posición(1.2216004070216808,22.163626186323366)
Proyectil en posición(1.3155696691002716,20.874828200655934)
Proyectil en posición(1.4095389311788624,19.4880302149885)
Proyectil en posición(1.5035081932574532,18.003232229321064)
Proyectil en posición(1.597477455336044,16.42043424365363)
Proyectil en posición(1.6914467174146348,14.739636257986197)
Proyectil en posición(1.7854159794932256,12.960838272318764)
Proyectil en posición(1.8793852415718164,11.08404028665133)
Proyectil en posición(1.9733545036504072,9.109242300983897)
Proyectil en posición(2.067323765728998,7.036444315316464)
Proyectil en posición(2.161293027807589,4.86564632964903)
Proyectil en posición(2.2552622898861796,2.5968483439815957)
Proyectil en posición(2.3492315519647704,0.2300503583141622)
Tras 26.000000000000007 intervalos de 0.1 (2.600000000000001 segundos) el proyectil
ha aterrizado.
Ha recorrido una distancia de 2.443200814043361
Ha alcanzado una altura máxima de 30 metros
```

Ejercicio 8

Definir una función mi_grep(cadena,fichero) similar al comando grep de unix (sin uso de patrones). Es decir, escribe por pantalla las líneas de fichero en las que ocurre cadena, junto con el número de línea.

In [28]:
```
def mi_grep(cadena, fichero):
    file = open(fichero, 'r')
    n = len(cadena)
    for line in file:
        index = line.find(cadena)
        if index != -1:
            print(line[:-1])
            print('{}{}'.format(' '*index, '^'*n))
    file.close()
```

In [29]:
```
# Usamos el soneto "Un soneto me manda hacer Violante" de Lope de Vega
# que hemos guardado en el fichero "soneto.txt"

mi_grep("verso","soneto.txt")

# Salida esperada:
# Línea 2: catorce versos dicen que es soneto;
#                  ^^^^^
# Línea 10: pues fin con este verso le voy dando.
#                             ^^^^^
# Línea 12: que voy los trece versos acabando;
#                             ^^^^^
```

```
catorce versos dicen que es soneto;
        ^^^^^
pues fin con este verso le voy dando.
                  ^^^^^
que voy los trece versos acabando;
                  ^^^^^
```

Ejercicio 9: Problema del viajante - Resolución por fuerza bruta

El objetivo de este ejercicio preliminar es constatar la dificultad de resolver el problema del viajante por fuerza bruta cuando aumenta el número de ciudades.

In [1]:
```
import random, time, math
from itertools import permutations
```

Primero definiremos clase `Viajante_n` que nos permita crear problemas del viajante con diferente numero de ciudades. Cada instancia de la clase dependerá de un valor n que indicará el número de ciudades y de un parámetro `escala` . Las coordenadas `x` e `y` de cada ciudad se tomaran aleatoriamente en el rango `[-escala,+escala]` .

¿Qué necesitamos en la clase viajante para representar y resolver el problema? ...

In [10]:
```
class Viajante_n():

    # Información necesaria:
    # - ciudaes
    # - coordenadas de ciudades
    # - ¿Qué estructura de datos es la más adecuada?
    def __init__(self, n, escala):
        # self.
        self.coordenadas = {}
        for i in range(n):
#             self.coordenadas[i] = (random.uniform(-escala,escala),random.uniform(-
            self.coordenadas[i] = (random.uniform(0,escala),random.uniform(0,escala)

    # Distancia Manhattan: ABS(x1-x2) + ABS(y1-y2)
    def distancia_manhattan(self, c1, c2):
        coord_c1 = self.coordenadas[c1]
        coord_c2 = self.coordenadas[c2]
        return abs(coord_c1[0]-coord_c2[0]) + abs(coord_c1[1]-coord_c2[1])

    # Distancia Euclidea: SQRT((x1-x2)^2 + (y1-y2)^2)
    # Raíz cuadrada: math.sqrt()
    def distancia_euclidea(self, c1, c2):
        coord_c1 = self.coordenadas[c1]
        coord_c2 = self.coordenadas[c2]
```

```
        return math.sqrt((coord_c1[0]-coord_c2[0])**2 + (coord_c1[1]-coord_c2[1])**2

    # Opcional: Personalizar la representación en texto de las instancias de la clas
    def __str__(self):
        ret = "Problema del viajante de {} ciudades.".format(len(self.coordenadas))
        for c, coords in self.coordenadas.items():
            ret += "\nLas coordenadas de la ciudad {} son {}:".format(c, coords)
        return ret

    # Ejemplo llamada función: distancia_circuito([0, 3, 1, 2])
    def distancia_circuito(self, circuito):
        ret = 0
        for i in range(len(circuito)-1):
            ret += self.distancia_euclidea(circuito[i], circuito[i+1])
        ret += self.distancia_euclidea(circuito[-1], circuito[0])
        return ret

    # Podemos simplificar la función anterior usando listas por comprensión.
    def distancia_circuito_lc(self, circuito):
        return sum([self.distancia_euclidea(i, i+1) for i in range(len(circuito)-1)]
```

A continuación definiremos la función `optimizacion_por_fuerza_bruta(pv)` que toma una instancia concreta de la clase `Viajante_n` , devuelve la ruta de mínima distancia, la distancia de esa ruta e imprime el tiempo (en segundos) necesario para realizar los cálculos.

¿En que consistiría una estrategia por fuerza bruta para resolver el problema?

In [13]:
```
# Para inicializar una variable con un número muy alto, podemos usar infinito (math.
def optimizacion_por_fuerza_bruta(pv):
    ti = time.time()
    posibles_rutas = permutations(pv.coordenadas.keys())
    menor_distancia = math.inf
    menor_ruta = None

    for ruta in posibles_rutas:
        d_ruta = pv.distancia_circuito(ruta)
        if d_ruta < menor_distancia:
            menor_distancia = d_ruta
            menor_ruta = ruta

    tf = time.time() - ti
    print("Tiempo empleado en encontrar la mejor ruta: {}".format(tf))
    print("Se han procesado {} rutas posibles".format(math.factorial(len(pv.coordena

    return menor_ruta, menor_distancia
```

Prueba con los siguientes ejemplos:

In [14]:
```
pv4=Viajante_n(4,10)
optimizacion_por_fuerza_bruta(pv4)
```

```
Tiempo empleado en encontrar la mejor ruta: 0.00020122528076171875
Se han procesado 24 rutas posibles
```

Out[14]: ((0, 1, 3, 2), 32.90192836044443)

In [15]:
```
pv6=Viajante_n(6,30)
optimizacion_por_fuerza_bruta(pv6)
```

```
Tiempo empleado en encontrar la mejor ruta: 0.007700204849243164
Se han procesado 720 rutas posibles
```

Out[15]: ((0, 4, 5, 3, 2, 1), 99.19100994993548)

In [16]:
```
pv8=Viajante_n(8,40)
optimizacion_por_fuerza_bruta(pv8)
```

```
Tiempo empleado en encontrar la mejor ruta: 0.24517059326171875
Se han procesado 40320 rutas posibles
```

Out[16]: ((0, 1, 3, 6, 4, 7, 2, 5), 244.09887560401538)

In [17]:
```
pv9=Viajante_n(9,40)
optimizacion_por_fuerza_bruta(pv9)
```

```
Tiempo empleado en encontrar la mejor ruta: 1.820465326309204
Se han procesado 362880 rutas posibles
```

Out[17]: ((1, 7, 2, 0, 4, 6, 5, 8, 3), 231.73415188095325)

Cuidado con las siguientes pruebas. Dependiendo de tu ordenador, la computación puede tardar bastante.

In [18]:
```
pv10=Viajante_n(10,50)
optimizacion_por_fuerza_bruta(pv10)
```

```
Tiempo empleado en encontrar la mejor ruta: 19.51688265800476
Se han procesado 3628800 rutas posibles
```

Out[18]: ((1, 0, 2, 9, 6, 5, 3, 7, 4, 8), 240.91220565999632)

In [ ]:
```
pv12=Viajante_n(12,60)
optimizacion_por_fuerza_bruta(pv12)
```