Tema 1:
Python 3

Índice

1. Control de errores: excepciones

Una excepción es un error que se produce durante la ejecución de un programa.

El control de excepciones es un mecanismo que permite realizar el tratamiento de estos errores. Por ejemplo en la introducción de datos erróneos que lleva a la realización de operaciones invalidas: divisiones por cero, uso de tipos inválidos en operaciones, …

Ejemplos de excepciones pueden ser:

```
5/0
  Traceback (most recent call last):
    File "/Codigo/excepciones.py", line 1, in <module>
      5/0
      ~^~
  ZeroDivisionError: division by zero
```

```
numeroa=int("cinco")
  File "/Codigo/excepciones.py", line 2, in <module>
      numeroa=int("cinco")
              ^^^^^^^^^^^^
  ValueError: invalid literal for int() with base 10: 'cinco'
```

Podemos ver en rojo el nombre de la excepción generado, que se podrá usar para tratar las excepciones y, al lado, un mensaje descriptivo de la excepción lanzada.

Se puede consultar una lista de excepciones predefinidas en Python en el siguiente enlace:

https://docs.python.org/es/3/library/exceptions.html#bltin-exceptions

1.1. Excepciones genéricas

Trata todas las posibles excepciones que pueden surgir sin importar su tipo. Su formato es el siguiente:

```
try:
    # Código que puede producir una excepción
except:
    # Qué hacemos si se produce
```

Si en el código ponemos un try obligatoriamente tiene que tener su correspondiente except (o finally que ya veremos).

Donde:

- Primero se ejecuta el código que hay entre el try y el except.
- Si no se produce una excepción el código dentro del except no se ejecuta.
- Si se produce el flujo del programa salta al except y se ejecuta su código.

Un ejemplo de código sería el siguiente:

```
try:
    edad = int(input("Introduce tu edad: "))
    if edad > 18:
        print("Eres mayor de edad")
    else:
        print("Eres menor de edad")
    viajes = int(input("¿Cuantos viajes has hecho en tu vida?: "))
    print("Has realizado", edad/viajes, "al año")

except:
    print("Debes introducir un número entero")
```

En este ejemplo except trata de forma conjunta los dos tipos de excepciones que se pueden producir: un ValueError si al método de conversión int() se le pasa un valor que no se pueda convertir a entero o un ZeroDivisionError si viajes tiene un valor 0.

1.2. Excepciones específicas

Las excepciones específicas buscan realizar un tratamiento individualizado para algún tipo de excepción en concreto.

Veamos el siguiente código:

```
try:
    edad = int(input("Introduce tu edad: "))
    if edad > 18:
        print("Eres mayor de edad")
    else:
        print("Eres menor de edad")
    viajes = int(input("¿Cuantos viajes has hecho en tu vida?: "))
    print("Has realizado", edad/viajes, "al año")

except ValueError:
    print("hay que introducir un entero")

except ZeroDivisionError:
    print("Las divisiones por cero no son válidas")

except:
    print("Por aquí ejecutaríamos el resto de excepciones")
```

Donde:

- Las excepciones se tratan por el orden en que se han definido y el programa entra por la primera excepción que coincida.
- except debe definirse al final ya que si la pusiésemos de primera siempre entraría por ella y no trataría el resto de excepciones.
- Es buena práctica ser lo más específico posible con los tipos de excepciones que pretendemos manejar

En un except podemos gestionar más de una excepción. Por ejemplo:

```
try:
    edad = int(input("Introduce tu edad: "))
    if edad > 18:
        print("Eres mayor de edad")
    else:
        print("Eres menor de edad")
    viajes = int(input("¿Cuantos viajes has hecho en tu vida?: "))
    print("Has realizado", edad/viajes, "al año")

except (ValueError, ZeroDivisionError):
    print("Valor no válido o división por cero ")

except:
    print("Por aquí ejecutaríamos el resto de excepciones")
```

1.3. Finally y else

Además de except en el tratamiento de excepciones podemos usar las dos siguientes clausulas opcionales:

- finally: se utiliza para ejecutar código al final del try ya se hayan producido excepciones o no.
- else: se utiliza para ejecutar código si no se han producido excepciones.

1.4. Información de las excepciones

Veamos el siguiente código:

```
try:
     edad = int(input("Introduce tu edad: "))
     edad=edad/0

except ValueError as fallo:
    print("hay que introducir un entero ", fallo.__str__())

except Exception as error:
    print("--------------------")
    print("Argumentos de la  de excepcion: ",error.args)
    print("Tipo de excepcion: ",type(error))    # the exception type
    print("Excepcion: ",error)
    print("Cadena con los argumentos: ",error.__str__())
```

Mediante la clausula **as** podemos obtener la excepción generada para obtener información de ella.

Excepción se puede usar como un comodín, que atrapa casi todas las excepciones y en este caso la usamos con la excepción genérica.

2. Listas

Las listas son estructuras de datos que permiten almacenar varios datos de forma lineal y ordenada. Pueden contener cualquier tipo en cada una de sus "casillas" incluso otras listas.

Para su definición se utilizan los corchetes [] y sus elementos se separan con comas **,** .

Los siguientes son ejemplos de su definición:

```
numeros = [ 1, 2, 4, 5, 8 ]
colores = [ 'azul', 'blanco', 'verde', 'blanco' ] # puede repetir valores
mezcla = [ 1, 'azul', 2, 'blanco', 3, 'verde' ] # puede mezclar tipos
vacia = [ ] # lista vacia que no tiene elementos
```

Si queremos imprimir todo el contenido de una lista podemos hacer:

```
print("Contenido de array:", numeros)
print("Contenido de matriz:", colores)
print("Contenido de lista:", mezcla)
```

2.1. Acceso a los elementos de una lista

Para acceder los elementos de una lista se utiliza su índice, esto es su posición dentro de la lista. Hay que tener el cuenta que la primera posición de una lista es la posición 0. Para acceder a una posición concreta se pone el nombre de las lista y entre corchetes la posición a la que queremos acceder.

Por ejemplo si tomamos como referencia la colores matriz (definida en el punto anterior):

Índice	0	1	2	3	Longitud (nº del elementos
Contenido	azul	blanco	verde	blanco	**4**

- matriz[0]: primer elemento de la lista → azul
- matriz[1]: segundo elemento de la lista → blanco
- matriz[2]: tercer elemento de la lista → verde
- matriz[4]: cuarto y último elemento de la lista → blanco

Para obtener la longitud de una lista, es decir el número de elementos que posee, podemos usar la función **len:**

```
print(len(matriz))
  4

print(len(vacia))
  0
```

Por lo que el último elemento de una lista sería:

```
print(matriz[len(matriz)-1]) # 4 - 1 = 3 puesto que los índices comienzan en 0
```

Si intentamos acceder a una posición que no existe se produce una excepción: **IndexError**.

Podemos usar también **índices negativos**, donde -1 es la última posición, -2 la penúltima, … y así sucesivamente.

```
print(matriz[-1])
  blanco

print(matriz[-2])
  verde
```

Si queremos modificar el valor de una posición de una lista asignamos una valor a esa posición, al igual que hacemos cuando asignamos un valor a una variable.

```
matriz[2]="amarillo"
print(matriz[2])
  amarillo
```

2.2. Pertenencia a una lista: in y no in

Para comprobar si un elemento pertenece a una lista tenemos:

- in: devuelve verdadero si el elemento pasado **pertenece** a la lista.
- not in: devuelve falso si el elemento pasado **no pertenece** a la lista.

Por ejemplo:

```
print("naranja" in colores)
  False

print("naranja" not in colores)
  True

if "naranja" in colores:
    print("El naranja pertenece a la lista")
else:
    print("El naranja no pertenece a la lista")
  El naranja no pertenece a la lista
```

2.3. Recorrer una lista

Para recorrer una lista podemos utilizar:

- Sus valores
- Sus índices

Para recorrer una lista por sus valores usamos un **for** junto el operador **in** y el nombre de la lista:

```
for color in colores:
    print(color, end=" ")
```
```
azul blanco verde blanco
```

Lo que hace este código es recorrer la lista copiando en cada iteración el valor de la lista correspondiente en la variable color.

Esta forma de recorrer la lista no permite modificar el contenido de esta ya que trabajamos con la copia de valor y no con el valor en sí mismo.

Si queremos modificar el contenido de una lista en un bucle tememos que recorrerlo mediante sus índices.

Por ejemplo:

```
for i in range(len(colores)):
    print(i, end=" ")
```
```
0 1 2 3
```

De esta forma obtenemos los índices de todos los elementos de la lista. Por lo que si queremos acceder a los elementos usaremos:

```
for i in range(len(colores)):
    print(colores[i], end=" ")
```
```
azul blanco verde blanco
```

Esta manera de recorrer una lista si permite modificar su contenido:

```
for i in range(len(colores)):
    if colores[i]=="blanco":
        colores[i]="amarillo"

for i in range(len(colores)):
    print(colores[i], end=" ")
```
```
azul amarillo verde amarillo
```

2.4. Slices

Mediante slices podemos acceder a los elementos de una lista usando rangos.

Acceden a un rango se hace de forma similar a acceder a una posición de la lista, pero en vez de indicar un índice se indica un rango con el formato inicio:fin del rango. Donde el inicio se incluye y el fin no.

```
lista[inicio_rango : fin_rango]
```

Los valores que devuelve se pueden ver desde dos puntos de vista:

- Muestra los valores desde la posición inicio hasta la posición fin teniendo en cuenta que fin no está incluido (muestra hasta el elemento anterior a fin).
- Muestra desde inicio (fin-inicio) valores.

Por ejemplo:

```
print(colores[1:4]) # desde la posición 1 hasta la posición anterior a 4, la 3
print(colores[1:4]) # desde la posición 1 muestra (4-1) = 3 elementos -> 1 2 3
```
```
['blanco', 'verde', 'blanco']
```

Si se omite el valor inicial comienza desde 0.

```
print(colores[:3])
```

es lo mismo que

```
print(colores[0:3])
  ['azul', 'blanco', 'verde']
```

De la misma forma si omitimos el final se muestra hasta el final.

```
print(colores[1:])
```

es lo mismo que

```
print(colores[1:len(colores)])
  ['blanco', 'verde', 'blanco']
```

Si queremos ver los dos últimos colores se puede hacer

```
print(colores[-2:])
  ['verde', 'blanco']
```

Si omitimos ambos valores es la lista completa:

```
print(colores[:])
```

es lo mismo que

```
print(colores[0:len(colores)])
  ['azul', 'blanco', 'verde', 'blanco']
```

Al igual que con los elementos individuales se pueden asignar rangos que incluso pueden cambiar el tamaño de la lista:

Por ejemplo:

```
colores = [ 'azul', 'blanco', 'verde', 'blanco' ]
colores[1:3]=["rojo","amarillo"]
print(colores)
  ['azul', 'rojo', 'amarillo', 'blanco']
```

En el que cambiamos el segundo elemento (blanco) por rojo y el tercero (verde) por amarillo.

Pero incluso:

```
colores = [ 'azul', 'blanco', 'verde', 'blanco' ]
colores[1:2]=["rojo","amarillo"]
print(colores)
  ['azul', 'rojo', 'amarillo', 'verde', 'blanco']
```

Donde se cambia la segunda posición, blanco. por lo colores rojo y amarillo

O borrar elementos:

```
colores = [ 'azul', 'blanco', 'verde', 'blanco' ]
colores[1:3]=["rojo"]
print(colores)
  ['azul', 'rojo', 'blanco']
```

Incluso vaciar la lista:

```
colores[:]=[]
```

Consideraciones a tener en cuenta

Imaginemos el siguiente código:

```
colores = [ 'azul', 'blanco', 'verde', 'blanco' ] # puede repetir valores
nuevosColores=colores

nuevosColores[1]="negro"

for color in colores:
    print(color)
```

¿Qué valores piensas que tiene ahora colores?

En Python cuando se hace una asignación de una estructura, como puede ser una lista, no se copian los valores de una lista en otra, sino que a la nueva variable se la asigna la dirección de memoria de la primera (en el ejemplo a nuevosColores se le asigna la dirección de memoria de colores). Por lo que a todos los efectos son la misma variable y si se modifica una se modifica la otra ya que son la misma.

En cambio slices devuelve una nueva lista por lo que:

```
colores = [ 'azul', 'blanco', 'verde', 'blanco' ] # puede repetir valores
nuevosColores=colores
nuevosColores2=colores[:]

nuevosColores[1]="negro"
nuevosColores2[2]="marron"

for color in colores:
    print(color, end=" ")
```

```
azul negro verde blanco
```

Donde:

- nuevosColores2=colores[:] → copia una lista en una lista nueva.
- nuevosColores[1]="negro" → modifica colores, ya que a todos los efectos son la misma lista.
- nuevosColores2[2]="marron" → no modifica colores, ya que una slice devuelve una lista nueva.

Si queremos comprobar si dos objetos son iguales tenemos la función id.

Si tenemos en cuenta los ejemplos anteriores:

```
nuevosColores=colores
nuevosColores2=colores[:]
```

Si queremos comprobar si el contenido de dos listas es igual podríamos hacer:

```
print(nuevosColores==colores)
```

```
True
```

```
print(nuevosColores2==colores)
```

```
True
```

Pero si queremos comprobar si dos listas son iguales, se cumple si el valor devuelto por la función id es el mismo:

```
print(id(nuevosColores)==id(colores))
```

```
True
```

```
print(id(nuevosColores2)==id(colores))
```

```
False
```

Saltos

En los rangos se puede poner un tercer argumento que indica el salto, cada cuantos elementos obtenemos:

```
lista[inicio_rango : fin_rango : salto]
```

Por ejemplo si queremos listar los elementos impares de los colores:

```
print(colores[0::2])
  ['azul', 'verde']
```

Si el salgo es negativo empezamos por el final. En este caso inicio_rango tiene que ser desde donde empezamos a recorrer la lista que en este caso es el final y fin_rango será donde se termina de recorrer la lista que será el inicio.

Por ejemplo:

```
print(colores[:1:-1])
  ['blanco', 'verde']
```

Desde el final recorremos hasta la posición 1, que no se incluye, por lo que lista el color que está en la posición 3 y el que está en la posición 2.

Si queremos invertir una lista podemos usar:

```
print(colores[::-1])
  ['blanco', 'verde', 'blanco', 'azul']
```

2.5. Eliminación de elementos

La instrucción **del** permite eliminar un elemento por su índice o rango. Para los ejemplos consideramos siempre la lista inicial, a la que no se le ha borrado ningún elemento.

Ejemplo de la eliminación de un elemento

```
del colores[1] # Se elimina segundo color, el primero tiene índice 0
print (colores)
   ['azul', 'verde', 'blanco']
```

Ejemplo de la eliminación de un rango de elementos:

```
del colores[1:3] # Se elimina segundo y tercer color
print (colores)
  ['azul', 'blanco']
```

Ejemplo de la eliminación de todos los elemento de la lista:

```
del colores[:]
print (colores)
  []
```

Si en vez de eliminar todos los elementos de la lista queremos eliminar la propia lista la pasamos a del la lista:

```
del colores
print (colores)
    print (colores)
           ^^^^^^^
  NameError: name 'colores' is not defined
```

Al eliminar la lista está deja de existir no pudiéndose referenciar.

2.6. Operador + en listas

Este operador permite unir dos listas en una.

```
numeros = [ 1, 2, 4, 5, 8 ]
colores = [ 'azul', 'blanco', 'verde', 'blanco' ] # puede repetir valores
nuevaLista=numeros + colores;
print(nuevaLista)
  [1, 2, 4, 5, 8, 'azul', 'blanco', 'verde', 'blanco']
```

2.7. Operador * en listas

Al igual que con cadenas este operador permite repetir un valor en la lista.

```
colores2 = [ 'azul', 'blanco' ] * 3 # crea una lista con los valores
print(colores2)                     # azul y blanco repetidos 3 veces
```

```
['azul', 'blanco', 'azul', 'blanco', 'azul', 'blanco']
```

2.8. Métodos para trabajar con listas

Las listas disponen de los siguientes métodos. Para los ejemplos usaremos la lista números:

```
numeros=[1, 2, 3, 4, 5, 6 ]
```

Se puede acceder a todos los métodos de lista mediante dir(list).

- append: permite añadir un elemento al final de la lista.

```
numeros.append(7)
print(numeros)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

Es equivalente a: numeros[len(numeros):]=[7]

- insert: inserta un elemento en una posición dada. El primer argumento es el índice del elemento anterior al que se va a insertar, y el segundo el valor a insertar:

  Para insertar al principio de la lista se usa:

```
numeros.insert(0,7)
print(numeros)
```

```
[7, 1, 2, 3, 4, 5, 6]
```

Para insertar en cualquier posición (en este caso en la posición 2):

```
numeros.insert(1,7)
print(numeros)
```

```
[1, 7, 2, 3, 4, 5, 6]
```

- remove: elimina el primer elemento de la lista cuyo valor es igual a un valor. Si no existe tal elemento, se produce un error ValueError.

```
numeros.remove(3)
print(numeros)
```

```
[1, 2, 4, 5, 6]
```

- clear: elimina todos los elementos de la lista. Equivale a del numeros[:]

```
numeros.clear()
print(numeros)
```

```
[]
```

- count: devuelve el número de veces que aparece un elemento en la lista.

```
print(numeros.count(2))
```

```
1
```

- copy: devuelve una copia de la lista

```
num=numeros.copy()
print(id(numeros)==id(num))
```

```
False
```

- reverse: invierte los elementos de la lista.

```
numeros.reverse()
print(numeros)
```

```
[6, 5, 4, 3, 2, 1]
```

- pop: **elimina** el elemento en la posición dada de la lista y lo devuelve. Si no se especifica ningún índice, numeros.pop() elimina y devuelve el último elemento de la lista. Si la lista está vacía o el índice está fuera del rango de la lista, se produce un error IndexError.

```
num=numeros.pop(2)
print(num)
   3
print(numeros)
   [1, 2, 4, 5, 6]
```

- index. Su formato es:

```
numeros.index(x [, inicio[, fin]])
```

Devuelve el índice del primer elemento cuyo valor es igual a x. Genera un error ValueError si no existe tal elemento.

Los argumentos opcionales inicio y fin establecen el principio y el final de la lista a buscar. El índice devuelto se calcula en relación al principio de la lista

```
print(numeros.index(5))
   4
print(numeros.index(5,2,6))
   4
```

- sort. Su formato es:

```
list.sort(key=None, reverse=False)
```

Ordena la lista y no devuelve nada (la lista queda ordenada). Para modificar la ordenación por defecto se puede usar los siguientes elementos:

- reverse: retorna la lista con un orden inverso:
- key: es una función que se utiliza como base para la ordenación.

Comando como ejemplo el siguiente ejemplo:

```
dias=["lunes", "Martes", "miércoles", "jueves", "Viernes"]
dias.sort();
print(dias)
   ['Martes', 'Viernes', 'jueves', 'lunes', 'miércoles']

dias.sort(reverse=True);
   ['miércoles', 'lunes', 'jueves', 'Viernes', 'Martes']

dias.sort(key=len, reverse=False);
   ['lunes', 'Martes', 'jueves', 'Viernes', 'miércoles']

dias.sort(key=str.lower);
   ['jueves', 'lunes', 'Martes', 'miércoles', 'Viernes']
```

Su uso es similar al de la función sorted. Mientras que sort solo se puede usar con listas sorted se puede utilizar con cualquier tipo de colección. Devuelve una colección ordenada dejando la original sin modificar.

- extend: añade nuevos objetos al final de la lista, a diferencia de append puede ser más de uno.

```
fin_semana=["sábado", "domingo"]
dias.extend(fin_semana)
   ['lunes', 'Martes', 'miércoles', 'jueves', 'Viernes', 'sábado', 'domingo']
```

2.9. Llenando de listas

Python ofrece la siguiente expresión para el llenado automático de listas:

```
<nueva_lista> = [<expresión> for <elemento> in <iterable>]
```

Que se traduce en:

```
<nueva_lista> = [<hacer_esto> para <cada_uno_de_los_elementos> de <esta_lista>]
```

Ejemplos de su utilización son los siguientes:

- Llenado de una lista con ceros:

```
matriz = [0 for i in range(5)]
print(matriz)
```
```
[0, 0, 0, 0, 0]
```

Es equivalente a:

```
matriz = []
for i in range(5):
 matriz.append(0)
print(matriz)
```
```
[0, 0, 0, 0, 0]
```

- Lista con los 10 primeros números

```
matriz = [i for i in range(1,11)]
print(matriz)
```
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Equivale a:

```
matriz = []
for i in range(1,11):
   matriz.append(i)
print(matriz)
```
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Lista con los números pares menores que 20

```
matriz = [i for i in range(2,20,2)]
print(matriz)
```
```
[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

- El doble de todos los elementos de una lista:

```
lista = [5, 8, 9, 6, 12, 11]
dobles = [2 * i for i in lista]
print(dobles)
```
```
[10, 16, 18, 12, 24, 22]
```

Equivale a:

```
lista = [5, 8, 9, 6, 12, 11]
dobles = []
for i in lista:
    dobles.append(2 * i)
print(dobles)
```
```
[10, 16, 18, 12, 24, 22]
```

Con condición

Se puede incluir una condición:

```
<nueva_lista> = [<expresión> for <elemento> in <iterable> if <condición>]
```

Solo se va a insertar si se cumple la condición:

Insertar lo elementos pares de la lista

```
lista = [1, 2, 3, 5, 6, 8, 11, 14]
pares = [x for x in lista if x % 2 == 0]
print(pares)
```

```
[2, 6, 8, 14]
```

2.10. Listas de bidimensionales

Python permite crear listas de lista con lo que se consigue crear listas de más de una dimensión:

Veamos el siguiente ejemplo:

```
dimesiones=[[1,2,3],[5,6,9],[3,7,4]]
```

Que podemos verlo de la siguiente forma:

```
dimesiones=[
              [1,2,3],
              [5,6,9],
              [3,7,4]
           ]
```

Cada elemento de la lista dimensiones es a su vez una lista.

Si queremos recorrer su contenido por filas:

```
for fila in dimesiones:
    for elemento in fila:
        print (elemento, end=' ')
    print() # nueva fila
```

En función de sus índices:

```
for x in range(len(dimesiones)):
    for y in range(len(dimesiones[x])):
        print(dimesiones[x][y], end=" ")
    print()
```

Imaginemos que sabemos que cada elemento de la primera lista tiene el mismo tamaño. Vamos a crear una función que muestra la lista pos sus columnas independientemente de su tamaño.

Si queremos recorrer su contendido por columnas:

```
# creamos una lista con un tamaño de 5x5
lista=[[1, 2, 3, 4, 5],
       [6, 7, 8, 9, 10],
       [11,12,13,14,15],
       [16,17,18,19,20],
       [21,22,23,24,25]
      ]

for j in range(5):
    for i in range(len(lista)):
        print(lista[i][j], end=" ")
    print()
```

```
1 6 11 16 21
2 7 12 17 22
```

```
3 8 13 18 23
4 9 14 19 24
5 10 15 20 25
```

Hay que tener en cuenta que pueden ser irregulares y que no todas las filas tienen que tener el mismo número de elementos.

3. Tuplas

Las tuplas son similares a las listas pero de tamaño fijo y son inmutables (una vez creada no se puede modificar). La ventaja es que son más ligeras con lo que se ahorra recursos y nos aseguramos que sus valores no van a ser modificados.

Las tuplas se crean con paréntesis, aunque también es posible crear una tupla separando los elementos con comas, sin paréntesis.

```
tupla_1 = (1, 2, 3)
print(tupla_1)
```

```
(1, 2, 3)
```

```
tupla_2 = 4, 5, 6
print(tupla_2)
```

```
(4, 5, 6)
```

Si se quiere crear una tupla de un solo elemento, se ha de indicar una coma al final, para diferenciarla de un valor individual.

```
tupla_individual = (1,)
print(tupla_individual)
```

```
(1,)
```

```
tupla_individual_2 = 2,
print(tupla_individual_2)
```

```
(2,)
```

```
tupla_cadenas = "Enero",
print(tupla_cadenas)
```

```
('Enero',)
```

Una tupla vacía se define de la siguiente manera:

```
tupla_vacia = ()
print(tupla_vacia)
```

```
()
```

Para acceder a la posición de una tupla se hace de la misma forma que con las listas. Lo mismo que para recorrerlas.

Como se comento las tuplas son inmutables. La única forma de modificar una tupla creada es asígnales otra tupla.

```
tupla = (1, 2, 3, 4, 5, 6)
print(tupla)
```

```
(1, 2, 3, 4, 5, 6)
```

```
tupla = (1, 2)
print(tupla)
```

```
(1, 2)
```

4. Random

Mediante el modulo random podemos generar números aleatorios.

Para su utilización necesitamos importar el módulo:

```
import random
```

Entre sus métodos tenemos:

- random.seed(): inicializamos el generador de números aleatorios. Si le pasamos un entero genera siempre la misma serie.

  ```
  random.seed(2)
  ```

- random.randint(*a*, *b*): devuelve un entero N que cumple `a <= N <= b`

  ```
  num=random.randint(0,10)
  ```

  Cada ejecución devuelve un valor aleatorio entre 0 y 10.

  Si se quiere forzar siempre la misma secuencia se utiliza seed.

  ```
  random.seed(1)
  print(random.randint(0,10))
  print(random.randint(0,10))
  ```

- random.random(): devuelve un float con valores entre `0.0 <= X < 1.0`
- random.uniform(*a*, *b*): devuelve un float con valores entre a `<= X < b` es lo mismo que aplicar la fórmula: `a + (b-a) * random()`
- random.shuffle(*x*): mezcla una secuencia de forma aleatoria.

  ```
  num=[1,2,3,4,5,6,7]
  random.shuffle(num)
  print(num)
  ```

  ```
  [2, 5, 4, 6, 7, 3, 1]
  ```

5. Ejercicios

1. Crea un programa que mientras el usuario no introduzca un número valido siga pidiendo entrada de datos.
2. Realiza un método al cual se le pasa una lista y devuelve la suma de sus elementos.
3. Crea una función que pida al usuario 5 números y los guarde en una lista y devuelva esa lista.
   - Muestra la lista
   - Calcula y muestra la media de los números
   - A continuación, reemplaza todos los números negativos de esa lista por 0.
   - Vuelve a mostrar la lista
4. A partir de una lista, escribe un programa que cree una lista nueva con los elementos de la primera lista que sean impares.
5. Crea una función que borre los números pares de una lista pasada como parámetro.
6. Escribe una función que reciba una lista de enteros y devuelva otra lista con los dobles de sus elementos.
7. Crea una lista con los nombres de los meses del año. Solicita al usuario que introduzca un nombre de mes. Si el nombre se encuentra en la lista, se mostrará el número de mes. Si el nombre no se encuentra en la lista, se mostrará un mensaje indicando que no existe ese mes, y se volverá a solicitar al usuario que introduzca un nombre de mes, hasta que introduzca el nombre de un mes válido.
8. Dada una lista de palabras, crea una nueva lista que contenga solo las palabras que tienen más de 5 letras.
9. A partir de una lista, escribe una función que cree una lista nueva eliminando aquellos que están repetidos.
10. A partir de una lista con los nombres de las personas que forman un grupo, solicita al usuario que introduzca un nombre y cuenta cuántas veces aparece dicho nombre en la lista.
11. Escribe una función que reciba dos listas y devuelva una tercera con los elementos comunes a ambas.
12. Utilizando una lista, escribe una función a la que se le pase el número del mes y devuelva su nombre.
13. Busca el algoritmo para calcular la letra del DNI y crea una función que use listas para devolver la letra del mismo.

    letras_dni = ["T", "R", "W", "A", "G", "M", "Y", "F", "P", "D", "X", "B", "N", "J", "Z", "S", "Q", "V", "H", "L", "C", "K", "E"]
14. Crea una función a la que se le pase un día de la semana y devuelva el siguiente, por ejemplo, si le pasamos lunes nos devolverá martes y si le pasamos domingo lunes.
15. Utilizando una tupla, escribe una función a la que se le pase el número del mes y devuelva su nombre.

16. Crea una función que devuelva la tabla de multiplicar del 0 al 10. Realiza su visualización para que se muestre de forma similar a la siguiente:

```
1       2       3       4       5       6       7       8       9       10
2       4       6       8       10      12      14      16      18      20
3       6       9       12      15      18      21      24      27      30
4       8       12      16      20      24      28      32      36      40
5       10      15      20      25      30      35      40      45      50
6       12      18      24      30      36      42      48      54      60
7       14      21      28      35      42      49      56      63      70
8       16      24      32      40      48      56      64      72      80
9       18      27      36      45      54      63      72      81      90
10      20      30      40      50      60      70      80      90      100
```

17. Crea una lista bidimensional que contenga los agrupamientos del alumnado de una clase para un determinado proyecto. La lista contendrá otras listas con los componentes de cada grupo.

    Muestra la lista por pantalla, con los componentes de cada grupo en una misma línea. Por ejemplo:

    Luis, Juan
    María, Elba, Mario
    Carola
    Pablo, Manuel, Raquel, Uxía

18. Utiliza una lista bidimensional para almacenar las temperaturas máximas de cada día de cada mes (como un número con un decimal). Cada mes (todos con 30 días) será una fila y las temperaturas estarán en columnas. Muestra las medias de cada mes. Para rellenar las temperaturas usa el random.

19. Crea una lista bidimensional de 6 filas y 7 columnas con valores aleatorios entre 0 y 9.
    - Muéstrala por pantalla.
    - Recórrela y almacena en una variable cuántos números pares hay en ella y almacena en otra cuántos impares. Muestra los resultados por pantalla.
    - Vuelve a recorrerla para crear una lista unidimensional nueva con los números pares que aparecen en ella y otra con los impares.

20. Crea una lista bidimensional 3x3 que sirva para jugar al tres en raya, almacenará 3 posibles cadenas, una cadena vacía para indicar que no se ha puesto una ficha en esa posición, una 'o' para indicar que ha jugado el jugador 1 y una 'x' para el jugador 2.
    - Crea una función que vacíe el tablero (rellene el tablero con cadenas vacías).
    - Crea una función que borre la pantalla y visualice el tablero. Por ejemplo:

```
-------
|o|x|o|
-------
|x|x|o|
-------
|x|o|x|
-------
```

    - Crea una función que la recorra y que busque si se ha producido 3 en raya para el jugador 1 en alguna de las filas.
    - Crea una función que la recorra y que busque si se ha producido 3 en raya para el jugador 1 en alguna de las columnas.
    - Crea una función que la recorra y que busque si se ha producido 3 en raya para el jugador 1 en alguna de las diagonales.
    - Crea una función que la recorra y que busque si se ha producido 3 en raya para algún jugador.
    - Completa el ejercicio para que dos jugadores puedan jugar al 3 en raya.

Llenando de listas

21. Crea una lista con las 10 primeras potencias de 3.
22. Crea una lista con los 100 primeros múltiplos de 7.
23. Crea una lista con los números que sean múltiplos de 3 o de 5 hasta el 100.
24. Crea una lista con los divisores de un número que se pasa por teclado.