Tipos de datos

Python

Literales

Literales enteros: 3, 12, -23
Literales reales: 12.3, 45.6
Literales cadenas:

```
'hola que tal!'
"Muy bien"
'''Podemos \n
ir al cine'''
```

- Los literales nos permiten representar valores.
- Estos valores pueden ser de diferentes tipos → diferentes tipos de literales:

  - **Literales numéricos**
    - Para representar números enteros utilizamos cifras enteras (Ejemplos: 3, 12, -23).
    - Para los números reales utilizamos un punto para separar la parte entera de la decimal (12.3, 45.6). Podemos indicar que la parte decimal es 0, por ejemplo 10., o la parte entera es 0, por ejemplo .001.

  - **Literales cadenas**

Nos permiten representar cadenas de caracteres.
Para delimitar las cadenas podemos usar el carácter ‘ o el carácter “.
También podemos utilizar la combinación “’ cuando la cadena ocupa más de una línea.
Ejemplos:

```
'hola que tal!’
"Muy bien“
'''Podemos \n
ir al cine'''
```

El caracter \n es el retorno de carro (los siguientes caracteres se escriben en una nueva línea).

Variables

- Una variables es un identificador que referencia a un valor. Para que una variable referencia a un valor se utiliza el operador de asignación =.

- El nombre de una variable, ha de empezar por una letra o por el carácter guión bajo, seguido de letras, números o guiones bajos.

```
>>> var = 5
>>> var
5
```

- python distingue entre mayúsculas y minúsculas en el nombre de una variable, pero se recomienda usar sólo minúsculas.

Expresiones

- Una expresión es una combinación de variables, literales, operadores, funciones y expresiones, que tras su evaluación o cálculo nos devuelven un valor de un determinado tipo.
- Ejemplos:

```
a + 7
(a ** 2) + b
```

Operadores

- Los operadores que podemos utilizar se clasifican según el tipo de datos con los que trabajen.
- Ejemplos:
  - Operadores aritméticos: +, -, *, /, //, %, **.
  - Operadores de cadenas: +, *
  - Operadores de asignación: =
  - Operadores de comparación: ==, !=, >=, >, <=, <
  - Operadores lógicos: and, or, not
  - Operadores de pertenencia: in, not in

Operadores- precedencia

1. Los paréntesis rompen la precedencia.
2. La potencia (**)
3. Operadores unarios (+ -)
4. Multiplicar, dividir, módulo y división entera (* % //)
5. Suma y resta (+ -)
6. Operador binario AND (&)
7. Operadores binario OR y XOR (^ |)
8. Operadores de comparación (<= < > >=)
9. Operadores de igualdad (<> == !=)
10. Operadores de asignación (=)
11. Operadores de pertenencia (in, in not)
12. Operadores lógicos (not, or, and)

Tipos de datos

- Numéricos
  - Tipo entero (int)
  - Tipo real (float)
- Booleanos (bool)
- Secuencia
  - Tipo lista (list)
  - Tipo tuplas (tuple)
- Cadenas de caracteres
  - Tipo cadena (str)
- Mapas o diccionario (dict)

Función *type()*

- La función *type()* nos devuelve el tipo de dato de un objeto dado.
- Ejemplos:

```
>>> type(5)
<class 'int'>
>>> type(5.5)
<class 'float'>
>>> type("hola")
<class 'str'>
>>> type([1,2])
<class 'list'>
```

Datos numéricos- TIPOS

- Enteros (int): Representan todos los números enteros (positivos, negativos y 0), sin parte decimal. En python3 este tipo no tiene limitación de espacio.
- Reales (float): Sirve para representar los números reales, tienen una parte decimal y otra decimal. Normalmente se utiliza para su implementación un tipo double de C.

```
>>> entero = 7
>>> type(entero)
<class 'int'>
>>> real = 7.2
>>> type (real)
<class 'float'
```

Operadores aritméticos

- +: Suma dos números
- -: Resta dos números
- *: Multiplica dos números
- /: Divide dos números, el resultado es float.
- //: División entera
- %: Módulo o resto de la división
- **: Potencia
- +, -: Operadores unarios positivo y negativo

Funciones predefinidas que trabajan con números

- abs(x): Devuelve al valor absoluto de un número.
- divmod(x,y): Toma como parámetro dos números, y devuelve una tubla con dos valores, la división entera, y el módulo o resto de la división.
- hex(x): Devuelve una cadena con la representación hexadecimal del número que recibe como parámetro.
- bin(x): Devuelve una cadena con la representación binaria del número que recibe como parámetro.
- pow(x,y): Devuelve la potencia de la base x elevado al exponente y. Es similar al operador**`.
- round(x,[y]): Devuelve un número real (float) que es el redondeo del número recibido como parámetro, podemos indicar un parámetro opcional que indica el número de decimales en el redondeo.

```
>>> abs(-7)
7
>>> divmod(7,2)
(3, 1)
>>> hex(255)
'0xff'
>>> pow(2,3)
8
>>> round(7.567,1)
7.6
```

Conversión de tipos

- int(x): Convierte el valor a entero.
- float(x): Convierte el valor a float.

```
>>> a=int(7.2)
>>> a
7
>>> type(a)
<class 'int'>
>>> a=int("345")
>>> a
345
>>> type(a)
<class 'int'>
>>> b=float(1)
>>> b
1.0
>>> type(b)
<class 'float'>
>>> b=float("1.234")
>>> b
1.234
>>> type(b)
<class 'float'>
```

- Si queremos convertir una cadena a entero, la cadena debe estar formada por caracteres numéricos, sino es así, obtenemos un error:

```
a=int("123.3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '123.3'
```

Otras operaciones

- En el módulo math encontramos muchas de estas operaciones. Para utilizarlas vamos a importar el módulo, por ejemplo para realizar una raíz cuadrada:

```
>>> import math
>>> math.sqrt(9)
3.0
```

Datos booleanos- TIPOS

- El tipo booleano o lógico se considera como un subtipo del tipo entero.
- Se puede representar dos valores: verdadero o falso (True, False).
- Cuando se evalúa una expresión, hay determinados valores que se interpretan como False:
  - False
  - Cualquier número 0. (0, 0.0)
  - Cualquier secuencia vacía ([], (), ‘’)
  - Cualquier diccionario vacío ({})

Operaciones de comparación

- Las expresiones lógicas utilizan operadores de comparación
- Permiten comparar dos valores y devuelven un valor booleano, dependiendo de lo que este comparando.
  - ==: Igual que
  - !=: Distinto que
  - >: Mayor que
  - <: Menor que
  - <=: Menor o igual
  - >=: Mayor o igual

Operadores booleanos o lógicos

- Los operadores booleanos se utilizan para operar sobre expresiones booleanas
- Se suelen utilizar en las estructuras de control alternativas (if, while):
  - x or y: Si x es falso entonces y, sino x. Este operados sólo evalúa el segundo argumento si el primero es False.
  - x and y: Si x es falso entonces x, sino y. Este operados sólo evalúa el segundo argumento si el primero es True.
  - not x: Si x es falso entonces True, sino False.

VARIABLE

- Una variables es un identificador que referencia a un valor.
- No hay que declarar la variable antes de usarla, el tipo de la variable será el mismo que el del valor al que hace referencia.
- El tipo de una variable puede cambiar en cualquier momento:

```
>>> var = 5
>>> type(var)
<class 'int'>
>>> var = "hola"
>>> type(var)
<class 'str'>
```

Creación, borrado y ámbito de variables

- Para crear una variable simplemente tenemos que utilizar un operador de asignación, el más utilizado = para que referencia un valor.
- Si queremos borrar la variable utilizamos la instrucción del.
- El ámbito de una variable se refiere a la zona del programa donde se ha definido y existe esa variable.
- Las variables creadas dentro de funciones o clases tienen un ámbito local, es decir no existen fuera de la función o clase.

```
>>> a = 5
>>> a
5
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Modificación del valor de una variable

- En cualquier momento podemos cambiar el valor de una variable, asignándole un nuevo valor:

```
>>> a = 5
>>> a
5
>>> a = 8
>>> a
8
```

- También podemos modificar el valor de una variable, por ejemplo si queremos incrementarla en uno, podríamos usar:

```
>>> a = a + 1
```

- Aunque también podemos utilizar otro operador de asignación:

```
>>> a+=1
```

- Otros operadores de asignación: +=, -=, *=, /=, %=, **=, //=.

FUNCION *INPUT*

- No permite leer por teclado información.
- Devuelve una cadena de caracteres y puede tener como argumento una cadena que se muestra en pantalla.
- Ejemplos:

```
>>> nombre=input("Nombre:")
Nombre:jose
>>> nombre
'jose'
>>> edad=int(input("Edad:"))
Edad:23
>>> edad
23
```

FUNCION *PRINT*

- No permite escribir en la salida estándar.
- Podemos indicar varios datos a imprimir, que por defecto serán separado por un espacio.
- Podemos también imprimir varias cadenas de texto utilizando la concatenación.

```
>>> print(1,2,3)
1 2 3

>>> print("Hola son las",6,"de la tarde")
Hola son las 6 de la tarde

>>> print("Hola son las "+str(6)+" de la tarde")
Hola son las 6 de la tarde
```

Formateo cadenas de caracteres

- Con la función *print* podemos indicar el formato con el que se va a mostrar los datos
- Ejemplo:

```
>>> print("%d %f %s" % (2.5,2.5,2.5))
2 2.500000 2.5

>>> print("El producto %s cantidad=%d precio=%.2f"%
("cesta",23,13.456))
El producto cesta cantidad=23 precio=13.46
```

CADENAS de Caracteres- Definición

- Las cadenas de caracteres (str): Me permiten guardar secuencias de caracteres.
- Podemos definir una cadena de caracteres de distintas formas:

```
>>> cad1 = "Hola"
>>> cad2 = '¿Qué tal?'
>>> cad3 = '''Hola,
    que tal?'''
```

Operaciones básicas

- **Concatenación**: +: El operador + me permite unir datos de tipos secuenciales, en este caso dos cadenas de caracteres.

```
>>> "hola " + "que tal"
'hola que tal'
```

- **Repetición**: *: El operador * me permite repetir un dato de un tipo secuencial, en este caso de cadenas de caracteres.

```
>>> "abc" * 3
'abcabcabc'
```

- **Indexación**: Puedo obtener el dato de una secuencia indicando la posición en la secuencia. En este caso puedo obtener el carácter de la cadena indicando la posición (empezando por la posición

```
>>> cadena = "josé"
>>> cadena[0]
'j'
>>> cadena[3]
'é'
```

- Para obtener la longitud de un caracteres que tiene), utilizamos la función len:

```
>>> cadena = "josé"
>>> len(cadena)
4
```

Comparación de cadenas

- Las cadenas se comparan carácter a carácter, en el momento en que dos caracteres no son iguales se compara alfabéticamente (es decir, se convierte a código unicode y se comparan).
- Ejemplos

```
>>> "a">"A"
True

>>> "informatica">"informacion"
True

>>> "abcde">"abcdef"
False
```

Estructuras de control

Python

Estructuras de control - Alternativa simple

- Al ejecutarse la instrucción if se evalúa la condición lógica.
  - Si la condición lógica es True se ejecutan de manera secuencial el bloque de instrucciones .
  - Si la condición es False no se ejecuta el bloque de instrucciones.
  - Una vez ejecutado el if (opción verdadera o falsa) se continúa la ejecución de forma secuencial por la siguiente instrucción (bloque de instrucción no identado).

```
edad = int(input("Dime tu edad:"))
if edad>=18:
    print("Eres mayor de edad")
print("Programa terminado")
```

Estructuras de control - Alternativa doble

- Al ejecutarse la instrucción if se evalúa la condición lógica.
  - Si la condición lógica es True se ejecutan de manera secuencial el primer bloque de instrucciones.
  - Si la condición es False se ejecuta el segundo bloque de instrucción.
  - Una vez ejecutado el if (opción verdadera o falsa) se continúa la ejecución de forma secuencial por la siguiente instrucción (bloque de instrucción no identado).

```
edad = int(input("Dime tu edad:"))
if edad>=18:
    print("Eres mayor de edad")
else:
    print("Eres menor de edad")
print("Programa terminado")
```

Estructuras de control - Alternativa múltiple

- En este caso tenemos varias opciones. Vamos preguntando por cada una de las opciones y según el valor de la expresión ejecutamos un bloque o otro.

```
nota = int(input("Dime tu nota:"))
if nota >=1 and nota <= 4:
    print("Suspenso")
elif nota == 5:
    print("Suficiente")
elif nota == 6 or nota == 7:
    print("Bien")
elif nota == 8:
    print("Notable")
elif nota ==9 or nota == 10:
    print("Sobresaliente")
else:
    print("Nota incorrecta")
print("Programa terminado")
```

Estructura repetitiva - While

- La instrucción while ejecuta una secuencia de instrucciones mientras una condición sea verdadera.

```
while <condición>:
    <instrucciones>
```

  - Al ejecutarse esta instrucción, la condición es evaluada. Si la condición resulta verdadera, se ejecuta una vez la secuencia de instrucciones que forman el cuerpo del ciclo. Al finalizar la ejecución del cuerpo del ciclo se vuelve a evaluar la condición y, si es verdadera, la ejecución se repite. Estos pasos se repiten mientras la condición sea verdadera.
  - Se puede dar la circunstancia que las instrucciones del bucle no se ejecuten nunca, si al evaluar por primera vez la condición resulta ser falsa.
  - Si la condición siempre es verdadera, al ejecutar esta instrucción se produce un ciclo infinito. A fin de evitarlo, las instrucciones del cuerpo del ciclo deben contener alguna instrucción que modifique la o las variables involucradas en la condición, de modo que ésta sea falsificada en algún momento y así finalice la ejecución del ciclo.

ejemplo

- Crea un programa que pida al usuario una contraseña, de forma repetitiva mientras que no introduzca “asdasd”. Cuando finalmente escriba la contraseña correcta, se le dirá “Bienvenido” y terminará el programa.

```
secreto = "asdasd"
clave = input("Dime la clave:")
while clave != secreto:
    print("Clave incorrecta!!!")
    clave = input("Dime la clave:")
print("Bienvenido!!!")
print("Programa terminado")
```

Instrucción *break*

- Termina la ejecución del bucle, además no ejecuta el bloque de instrucciones.
- Ejemplo

```
secreto = "asdasd"
clave = input("Dime la clave:")
while clave != secreto:
    print("Clave incorrecta!!!")
    otra = input("¿Quieres introducir otra clave (S/N)?:")
    if otra.upper()=="N":
        break;
    clave = input("Dime la clave:")
if clave == secreto:
    print("Bienvenido!!!")
print("Programa terminado")
```

Instrucción *continue*

- Deja de ejecutar las restantes instrucciones del bucle y vuelve a iterar.
- Ejemplo: usamos la instrucción *continue* para mostrar los números pares del 1 al 10:

```
cont = 0
while cont<10:
    cont = cont + 1
    if cont % 2 != 0:
        continue
    print(cont)
```

Instrucción *repetir – hasta que*

- En Python podemos simular el comportamiento de la estructura *repetir-hasta que* utilizando un bucle *while* con la instrucción *break*:

```
secreto = "asdasd"
while True:
    clave = input("Dime la clave:")
    if clave != secreto:
        print("Clave incorrecta!!!")
    if clave == secreto:
        break;
print("Bienvenido!!!")
print("Programa terminado")
```

Estructura repetitiva - for

- La estructura *for* nos permite recorrer los elementos de una secuencia (lista, tupla, cadena de caracteres,…).
- Vamos a usar *for* de forma similar a la instrucción *Para* de pseudocódigo, es decir, ejecutar una secuencia de instrucciones un número determinado de veces, desde un valor inicial, hasta un valor final y con un posible incremento.
  - Para ello vamos a usar el tipo de datos *range* que nos permite generar listas de números. Vamos a usar *for* para crear bucles de instrucciones donde sabemos a priori el número de iteraciones que hay que realizar.

ejemplos

- Escribir en pantalla del 1 al 10

```
for var in range(1,11):
    print(var," ",end="")
```

- Escribir en pantalla del 10 al 1

```
for var in range(10,0,-1):
    print(var," ",end="")
```

- Escribir los número pares desde el 2 al 10

```
for var in range(2,11,2):
    print(var," ",end="")
```

Variables- Contador

- Un **contador** es una variable entera que la utilizamos para contar cuando ocurre un suceso.

- Se inicializa a un valor inicial.

  cont = 0;

- Se incrementa, cuando ocurre el suceso que estamos contado se le suma 1.

  cont = cont + 1;

- Otra forma de incrementar el contador:

  cont += 1

- Ejemplo. Introducir 5 números y contar los pares

```
cont = 0;
for var in range(1,6):
    num = int(input("Dime un número:"))
    if num % 2 == 0:
        cont = cont + 1
print("Has introducido ",cont," números pares.")
```

Variables- Acumulador

- Un **acumulador** es una variable numérica que permite ir acumulando operaciones. Permite ir haciendo operaciones parciales.
- Se inicializa a un valor inicial según la operación que se va a acumular:
  - a 0 si es una suma
  - a 1 si es un producto.
- Se acumula un valor intermedio. acum = acum + num;

- Ejemplo. Introducir 5 números y sumar los pares

```
suma = 0;
for var in range(1,6):
    num = int(input("Dime un número:"))
    if num % 2 == 0:
        suma = suma + num
print("La suma de los números pares es ",suma)
```

Variables- Indicador

- Un **indicador** es una variable lógica, que usamos para recordar o indicar algún suceso.

- Se inicializa a un valor lógico que indica que el suceso no ha ocurrido.

  indicador = False

- Cuando ocurre el suceso que queremos recordar cambiamos su valor.

  indicador = True

- Ejemplo. Introducir 5 números e indicar si se ha introducido algún número par

```
indicador  =  False;
for var in range(1,6):
    num = int(input("Dime un número:"))
    if num % 2 == 0:
        indicador  = True
if indicador:
    print("Has introducido algún número par")
else:
    print("No has introducido algún número par")
```

Secuencias

Tipo de datos

Cadenas de caracteres

- Las cadenas de caracteres (str) permiten guardar secuencias de caracteres.

- Además de las operaciones:
  - Concatenación: +: El operador + me permite unir datos de tipos secuenciales, en este caso dos cadenas de caracteres.
  - Repetición: *: El operador * me permite repetir un dato de un tipo secuencial, en este caso de cadenas de caracteres.
  - Indexación: Puedo obtener el dato de una secuencia indicando la posición en la secuencia. En este caso puedo obtener el carácter de la cadena indicando la posición (**empezando por la posición 0**).
- Para obtener la longitud de una cadena (número de caracteres que tiene), utilizamos la función len.

Operaciones

```
>>> cadena = "informática"
>>> for caracter in cadena:
...     print(caracter,end="")
...
informática
```

- Las cadenas de caracteres se pueden recorrer:
- Operadores de pertenencia: Se puede comprobar si un elemento (subcadena) pertenece o no a una cadena de caracteres con los operadores in y not in.

```
>>> "a" in cadena
True
>>> "b" in cadena
False
>>> "a" not in cadena
False
```

- Slice (rebanada): Puedo obtener una subcadena de la cadena de caracteres. Se indica el carácter inicial, y el carácter final, además podemos indicar opcionalmente un salto. Si no se indica el carácter inicial se supone que es desde el primero, sino se indica el carácter final se supone que es hasta el final. Por último podemos usar salto negativo para empezar a contar desde el final.

Mas operaciones

- cadena[start:end] # Elementos desde la posición start hasta end-1
- cadena[start:] # Elementos desde la posición start hasta el final
- cadena[:end] # Elementos desde el principio hasta la posición end-1
- cadena[:] # Todos Los elementos
- cadena[start:end:step] # Igual que el anterior pero dando step saltos.

- Ejemplos:

```
>>> cadena[2:5]
'for'
>>> cadena[2:7:2]
'frá'
>>> cadena[:5]
'infor'
>>> cadena[5:]
'mática'
>>> cadena[-1:-3]
''
>>> cadena[::-1]
'acitámrofni'
```

Conversión de tipos

- Podemos convertir cualquier número en una cadena de caracteres utilizando la función str:

```
>>> cad = str(7.8)
>>> type(cad)
<class 'str'>
>>> print(cad)
7.8
```

Propiedad – inmutabilidad (I)

- Cuando creamos una variable de tipo cadena de caracteres, estamos creando un objeto de la clase str.
- Cada vez que creamos una variable de una determinada clase, creamos un objeto, que además de guardar información (en nuestro caso los caracteres de la cadena) puede realizar distintas operaciones que llamamos métodos.
- Ejemplo: el método upper() nos permite convertir la cadena a mayúsculas.

Propiedad – inmutabilidad (II)

- No podemos cambiar los caracteres de una cadena de la siguiente forma:

```
>>> cadena = "informática"
>>> cadena[2]="g"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- Esto implica que al usar un método la cadena original no cambia, el método devuelve otra cadena modificada. Ejemplo:

```
>>> cadena = "informática"
>>> cadena.upper()
'INFORMÁTICA'
>>> cadena
'informática'
```

- Si queremos cambiar la cadena debemos modificar su valor con el operador de asignación:

```
>>> cadena = cadena.upper()
>>> cadena
'INFORMÁTICA'
```

Métodos de formato

- capitalize() nos permite devolver la cadena con el primer carácter en mayúsculas.
- lower() y upper() convierte la cadena de caracteres en minúsculas y mayúsculas respectivamente.
- swapcase(): devuelve una cadena nueva con las minúsculas convertidas a mayúsculas y viceversa.
- title(): Devuelve una cadena con los primeros caracteres en mayúsculas de cada palabra.

- Ejemplos

```
>>> cad = "hola, como estás?"
>>> print(cad.capitalize())
Hola, como estás?
```

```
>>> cad = "Hola Mundo"
>>> print(cad.lower())
hola mundo

>>> cad = "hola mundo"
>>> print(cad.upper())
HOLA MUNDO
```

```
>>> cad = "Hola Mundo"
>>> print(cad.swapcase())
hOLA mUNDO
```

```
>>> cad = "hola mundo"
>>> print(cad.title())
Hola Mundo
```

Métodos de búsqueda

- count(): Es un método al que indicamos como parámetro una subcadena y cuenta cuantas apariciones hay de esa subcadena en la cadena.
- Además podemos indicar otro parámetro para indicar la posición desde la que queremos iniciar la búsqueda. Y otro parámetro optativo para indicar la posición final de búsqueda.
- find() nos devuelve la posición de la subcadena que hemos indicado como parámetro. Sino se encuentra se devuelve -1.

- Ejemplos

```
>>> cad = "bienvenido a mi aplicación"
>>> cad.count("a")
3
```

```
>>> cad.count("a",16)
2
>>> cad.count("a",10,16)
1
```

```
>>> cad.find("mi")
13
>>> cad.find("hola")
-1
```

Métodos de validación

- startswith() nos indica con un valor lógico si la cadena empieza por la subcadena que hemos indicado como parámetro. Podemos indicar también con otro parámetro la posición donde tiene que buscar.
- endswith() igual que la anterior pero indica si la cadena termina con la subcadena indicada. En este caso, se puede indicar la posición de inicio y final de búsqueda.
- Otras funciones de validación: isdigit(), islower(), isupper(), isspace(), istitle(),…

- Ejemplos

```
>>> cad.startswith("b")
True
>>> cad.startswith("m")
False
>>> cad.startswith("m",13)
True
```

```
>>> cad.endswith("ción")
True
>>> cad.endswith("ción",0,10)
False
>>> cad.endswith("nido",0,10)
True
```

Métodos de sustitución

- replace(): Devuelve una cadena donde se ha sustituido las apariciones de la primera subcadena indicada por la segunda subcadena indicada como parámetro.

```
>>> buscar = "nombre apellido"
>>> reemplazar_por = "Juan Pérez"
>>> print ("Estimado Sr. nombre apellido:".replace(buscar, reemplazar_por))
Estimado Sr. Juan Pérez:
```

- strip(): Devuelve una cadena donde se han quitado los espacios del principio y del final. Si indicamos una subcadena como parámetro quitará dicha subcadena del principio y del final.

```
>>> cadena = "   www.eugeniabahit.com
>>> print(cadena.strip())
www.eugeniabahit.com
>>> cadena="00000000123000000000"
>>> print(cadena.strip("0"))
123
```

Métodos de unión y división

- el método split() nos permite convertir una cadena en una lista.

```
>>> hora = "12:23:12"
>>> print(hora.split(":"))
['12', '23', '12']
```

- splitlines(): Nos permite separar las líneas que hay en una cadena (indicada con el carácter \n) en una lista.

```
>>> texto = "Linea 1\nLinea 2\nLinea 3"
>>> print(texto.splitlines())
['Linea 1', 'Linea 2', 'Linea 3']
```

Listas

Python

Introducción

- Si queremos guardar un conjunto de valores, en pseudocódigo utilizamos los arreglos (*arrays*). Un array (o arreglo) es una estructura de datos con elementos homogéneos, del mismo tipo, numérico o alfanumérico, reconocidos por un nombre en común.

- Hay muchos lenguajes que implementan los arrays, pero esta estructura tiene dos limitaciones: son homogéneas, es decir sólo se pueden guardar datos del mismo tipo, y son estáticas, a la hora de declarar se indican las posiciones y la longitud del array no se puede cambiar durante la ejecución del programa.

- En **Python** no existen los arrays, tenemos varios tipos de datos que nos permiten guardar conjuntos de informaciones. Las listas (list) permiten guardar un conjunto de datos que se pueden repetir y que pueden ser de distintos tipos. Además esta estructura es dinámica, en cualquier momento de la ejecución del programa puedo añadir o eliminar elementos de la lista.

Construcción de una lista

- Para crear una lista usamos los caracteres [y]:

```
>>> lista1 = []
>>> lista2 = ["a",1,True]
```

- Las listas son secuencias, a las que podemos realizar distintas operaciones. Vamos a ver varios ejemplos partiendo de la siguiente lista:

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

Operaciones básicas (I)

- Recorre listas:

```
>>> for num in lista:
...   print(num,end="")
123456
```

- Con la instrucción for podemos recorrer más de una lista, utilizando la función zip. Ejemplo:

```
>>> lista2 = ["a","b","c","d","e"]
>>> for num,letra in zip(lista,lista2):
...     print(num,letra)
...
1 a
2 b
3 c
4 d
5 e
```

Operaciones básicas (II)

- **Operadores de pertenencia**: Se puede comprobar si un elemento pertenece o no a una lista con los operadores in y not in.
- **Concatenación**: El operador + me permite unir datos de tipos listas
- **Repetición**: El operador * me permite repetir un dato de una lista

```
>>> 2 in lista
True
>>> 8 not in lista
True
```

```
>>> lista + [7,8,9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> lista * 2
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
```

Operaciones básicas (III)

- **Indexación**: Puedo obtener el dato de una secuencia indicando la posición en la secuencia.

```
>>> lista[3]
4
```

- Cada elemento tiene un índice, empezamos a contar por el elemento en el **índice 0**. Si intento acceder a un índice que corresponda a un elemento que no existe obtenemos una excepción IndexError.

```
>>> lista1[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module
IndexError: list index out of range
```

- Se pueden utilizar índices negativos

```
>>> lista[-1]
6
```

Operaciones básicas (IV)

- Slice (rebanada): Puedo obtener una subsecuencia de los datos de una lista.
- Funciona de forma similar como en las cadenas

- Ejemplos

```
>>> lista=[1,2,3,4,5,6]
>>> lista[2:4]
[3, 4]
>>> lista[1:4:2]
[2, 4]
>>> lista[:5]
[1, 2, 3, 4, 5]
>>> lista[5:]
[6]
>>> lista[::-1]
[6, 5, 4, 3, 2, 1]
```

Funciones predefinidas

```
>>> lista1=[20,40,10,40,50]
>>> len(lista1)
5
>>> max(lista1)
50
>>> min(lista1)
10
>>> sum(lista1)
160
>>> sorted(lista1)
[10, 20, 40, 40, 50]
>>> sorted(lista1,reverse=True)
[50, 40, 40, 20, 10]
```

Listas multidimensionales

- A la hora de definir las listas hemos indicado que podemos guardar en ellas datos de cualquier tipo, y evidentemente podemos guardar listas dentro de listas.

```
>>> tabla = [[1,2,3],[4,5,6],[7,8,9]]
>>> tabla[1][1]
5

>>> for fila in tabla:
...   for elem in fila:
...      print(elem,end="")
...   print()

123
456
789
```

Listas mutables

- Al igual que las cadenas, el tipo de datos lista es una *clase.*
- Cada vez que creamos una variable de la clase lista estamos creando un objeto que además de guardar un conjunto de datos, posee un conjunto de métodos que nos permiten trabajar con la lista.
- Los elementos de las listas se pueden modificar

```
>>> lista1 = [1,2,3]
>>> lista1[2]=4
>>> lista1
[1, 2, 4]
>>> del lista1[2]
>>> lista1
[1, 2]
```

- Esto también ocurre cuando usamos los métodos, es decir, los métodos de las listas modifican el contenido de la lista. Ej:

```
>>> lista1.append(3)
>>> lista1
[1, 2, 3]
```

Copiar listas

- Para copiar una lista en otra no podemos utilizar el operador de asignación:

```
>>> lista1 = [1,2,3]
>>> lista2 = lista1
>>> lista1[1] = 10
>>> lista2
[1, 10, 3]
```

- El operador de asignación no crea una nueva lista, sino que nombra con dos nombres distintos a la misma lista, por lo tanto la forma más fácil de copiar una lista en otra es:

```
>>> lista1=[1,2,3]
>>> lista2=lista1[:]
>>> lista1[1]=10
>>> lista1
[1, 10, 3]
>>> lista2
[1, 2, 3]
```

Métodos de inserción

- append(): añade un elemento a la lista:

```
>>> lista = [1,2,3]
>>> lista.append(4)
>>> lista
[1, 2, 3, 4]
```

- extend(): une dos listas:

```
>>> lista2 = [5,6]
>>> lista.extend(lista2)
>>> lista
[1, 2, 3, 4, 5, 6]
```

- insert(): añade un elemento en un posición indicada de la lista:

```
>>> lista.insert(1,100)
>>> lista
[1, 100, 2, 3, 4, 5, 6]
```

Métodos de eliminación

- pop(): elimina un elemento de la lista y lo devuelve. Se puede indicar el índice del elemento que queremos obtener como parámetro. Si no se indica se devuelve y elimina el último:

```
>>> lista.pop()
6
>>> lista
[1, 100, 2, 3, 4, 5]

>>> lista.pop(1)
100
>>> lista
[1, 2, 3, 4, 5]
```

- remove(): Elimina el elemento de la lista:

```
>>> lista.remove(3)
>>> lista
[1, 2, 4, 5]
```

Métodos de ordenación

- reverse(): modifica la lista invirtiendo los elementos:

```
>>> lista.reverse()
>>> lista
[5, 4, 2, 1]
```

- sort(): modifica la lista ordenando los elementos, se puede indicar el sentido de la ordenación

```
>>> lista.sort()
>>> lista
[1, 2, 4, 5]

>>> lista.sort(reverse=True)
>>> lista
[5, 4, 2, 1]

>>> lista=["hola","que","tal","Hola","Que","Tal"]
>>> lista.sort()
>>> lista
['Hola', 'Que', 'Tal', 'hola', 'que', 'tal']
```

Métodos de búsqueda

- count(): devuelve el número de apariciones de un elemento en la lista

```
>>> lista.count(5)
1
```

- index(): Nos devuelve la posición de la primera aparición del elemento indicado. Se puede indicar la posición inicial y final de búsqueda. Si no encuentra el elemento nos da una **excepción.**

```
>>> lista.append(5)
>>> lista
[5, 4, 2, 1, 5]
>>> lista.index(5)
0
>>> lista.index(5,1)
4
>>> lista.index(5,1,4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 5 is not in list
```

Tuplas (I)

- Las tuplas (tuple): sirven para los mismo que las listas, permiten guardar un conjunto de datos que se pueden repetir y que pueden ser de distintos tipos.
- Para crear una lista se usan los caracteres (y)

```
>>> tupla1 = ()
>>> tupla2 = ("a",1,True)
```

- Las tuplas son **inmutables**

```
>>> tupla = (1,2,3)
>>> tupla[1]=5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tuplas (II)

- Operaciones básicas:
  - Las tuplas se pueden recorrer
  - Operadores de pertenencia: in y not in
  - Concatenación: +
  - Repetición: *
  - Indexación
  - Slide
- Funciones
  - len
  - max
  - min
  - sum
  - sorted

- Métodos principales: count, index

```
>>> tupla = (1,2,3,4,1,2,3)
>>> tupla.count(1)
2

>>> tupla.index(2)
1
>>> tupla.index(2,2)
5
```

DICCIONARIOS

Tipos de datos

Definición

- Los diccionarios son tipos de datos que nos permiten guardar valores, a los que se puede acceder por medio de una clave (una cadena de caracteres o un número). Son tipos de datos mutables y los campos no tienen asignado orden.

```
>>> dicccionario = {'one': 1, 'two': 2, 'three': 3}
```

- Si tenemos un diccionario vacío, al ser un objeto mutable, también podemos construir el diccionario de la siguiente manera.

```
>>> dict1 = {}
>>> dict1["one"]=1
>>> dict1["two"]=2
>>> dict1["three"]=3
```

Operaciones básicas

- len(): devuelve número de elementos del diccionario
- **Indexación**: podemos obtener el valor de un campo o cambiarlo (si no existe el campo nos da una excepción KeyError)
- del():podemos eliminar un elemento (si no existe el campo nos da una excepción KeyError)
- Operadores de pertenencia: *key in d* y *key not in d*

- Ejemplo

```
>>> a = {'one': 1, 'two': 2, 'three': 3}
```

```
>>> len(a)
3
```

```
>>> a["one"]
1
>>> a["one"]+=1
>>> a
{'three': 3, 'one': 2, 'two': 2}
```

```
>>> del(a["one"])
>>> a
{'three': 3, 'two': 2}
```

```
>>> "two" in a
True
```

Característica – tipo mutable

- Los diccionarios, al igual que las listas, son tipos de datos mutable.

```
>>> a = {'one': 1, 'two': 2, 'three': 3}
>>> a["one"]=2
>>> del(a["three"])
>>> a
{'one': 2, 'two': 2}


>>> a = {'one': 1, 'two': 2, 'three': 3}
>>> b = a
>>> del(a["one"])
>>> b
{'three': 3, 'two': 2}
```

- Para copiar diccionarios, usamos el método copy():

```
>>> a = dict(one=1, two=2, three=3)
>>> b = a.copy()
>>> a["one"]=1000
>>> b
{'three': 3, 'one': 1, 'two': 2}
```

Método de eliminación

- clear(): elimina los elementos de un diccionario.

```
>>> dict1 = {'one': 1, 'two': 2, 'three': 3}
>>> dict1.clear()
>>> dict1
{}
```

Métodos de agregado y creación

- copy(): permite copiar diccionarios.

```
>>> dict1 = {'one': 1, 'two': 2, 'three': 3}
>>> dict2 = dict1.copy()
```

- update(): permite añadir elementos a un diccionario a partir de los elementos de otro diccionario.

```
>>> dict1 = {'one': 1, 'two': 2, 'three': 3}
>>> dict2 = {'four':4,'five':5}
>>> dict1.update(dict2)
>>> dict1
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5}
```

Métodos de retorno

- get(): devuelve el valor de un elemento de un diccionario indicando la clave. Además podemos indicar el valor devuelto si no existe el elemento.

```
>>> dict1 = {'one': 1, 'two': 2, 'three': 3}
>>> dict1.get("one")
1
>>> dict1.get("four")
>>> dict1.get("four","no existe")
'no existe'
```

- pop(): devuelve el valor de un elemento de un diccionario a partir de la clave y elimina dicho elemento. También podemos indicar el valor devuelto si no existe el elemento buscado.

```
>>> dict1.pop("one")
1
>>> dict1
{'two': 2, 'three': 3}
>>> dict1.pop("four")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'four'
>>> dict1.pop("four","no existe")
'no existe'
```

Recorrido de diccionarios

- El método keys() devuelve las claves de un diccionario, por lo que podemos recorrer las claves de la siguiente manera:

```
>>> for clave in dict1.keys():
...     print(clave)
one
two
three
```

- El método values() devuelve los valores de un diccionario, por lo que podemos recorrer los valores de la siguiente manera:

```
>>> for valor in dict1.values():
...     print(valor)
1
2
3
```

- El método items() devuelve la clave y el valor correspondiente de los elemento de un diccionario, por lo que podemos recorrer ambos:

```
>>> for clave,valor in dict1.items():
...     print(clave,"->",valor)
one -> 1
two -> 2
three -> 3
```

EXCEPCIONES

Errores sintácticos

- Los errores sintácticos se producen cuando tenemos algo mal escrito en nuestro código fuente. Ejemplo:

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                     ^
SyntaxError: invalid syntax
```

Error de ejecución

- Una excepción o un error de ejecución se produce durante la ejecución del programa. Las excepciones se puede manejar para que no termine el programa.
- Ejemplos:
  - ZeroDivisionError
  - NameError
  - TypeError

```
>>> 4/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero

>>> a+4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined

>>> "2"+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Manejo excepciones (I)

1. Se ejecuta el bloque de instrucciones de try.
2. Si no se produce la excepción, el bloque de except no se ejecuta y continúa la ejecución secuencia.
3. Si se produce una excepción, el resto del bloque try no se ejecuta, si la excepción que se ha produce corresponde con la indicada en except se salta a ejecutar el bloque de instrucciones except.
4. Si la excepción producida no se corresponde con las indicadas en except se pasa a otra instrucción try, si finalmente no hay un manejador nos dará un error y el programa terminará.

```
>>> while True:
...   try:
...     x = int(input("Introduce un número:"))
...     break
...   except ValueError:
...     print ("Debes introducir un número")
```

Manejo excepciones (II)

- Un bloque except puede manejar varios tipos de excepciones, si quiero controlar varios tipos de excepciones puedo poner varios bloques except. Teniendo en cuenta que en el último, si quiero no indico el tipo de excepción:

```
>>> try:
...   print (10/int(cad))
... except ValueError:
...   print("No se puede convertir a entero")
... except ZeroDivisionError:
...   print("No se puede dividir por cero")
... except:
...   print("Otro error")
```

- Se puede utilizar también la clausula else:

```
>>> try:
...   print (10/int(cad))
... except ValueError:
...   print("No se puede convertir a entero")
... except ZeroDivisionError:
...   print("No se puede dividir por cero")
... else:
...   print("Otro error")
```

Manejo excepciones (III)

- Podemos indicar una clausula finally para indicar un bloque de instrucciones que siempre se debe ejecutar, independientemente de la excepción se haya producido o no.

```
>>> try:
...     result = x / y
... except ZeroDivisionError:
...     print("División por cero!")
... else:
...     print("El resultado es", result)
... finally:
...     print("Terminamos el programa")
```

MÓDULOS

Definición

Módulo: Cada uno de los ficheros .py que nosotros creamos se llama módulo.

Los elementos creados en un módulo (funciones, clases, …) se pueden importar para ser utilizados en otro módulo. El nombre que vamos a utilizar para importar un módulo es el nombre del fichero.

Python tiene sus propios módulos, los cuales forman parte de su librería de módulos estándar, que podemos usar en nuestros programas. Para poder usarlos debemos importarlos.

Importación de módulos

- Podemos importar el módulo completo y posteriormente usar cualquier función definida, por ejemplo para realizar la raíz cuadrada importamos el módulo de funciones matemáticas math:

```
>>> import math
>>> math.sqrt(9)
3.0
```

- Si sólo queremos utilizar la función sqrt se puede importar sólo esa función (en este caso al utilizarla no hace falta utilizar el nombre del módulo):

```
>>> from math import sqrt
>>> sqrt(9)
3.0
```

Módulos matemáticos de Python

- Módulo **math**: El módulo math nos proporciones distintas funciones y operaciones matemáticas.

```
>>> math.sqrt(9)
```

- Módulo **fractions**: El módulo fractions nos permite trabajar con fracciones.

- Módulo **random**: El módulo random nos permite generar datos pseudo-aleatorios.

```
>>> random.randint(1,10)
```

Más módulos de Python

- Módulos de hora y fecha
  - Módulo time: permite trabajar con fechas y horas. El tiempo es medido como un número real que representa los segundos transcurridos desde el 1 de enero de 1970.

```
>>> time.sleep(1)
```

  - Módulo datetime: amplía las posibilidades del módulo time que provee funciones para manipular expresiones de tiempo.

- Módulo del sistema
  - Módulo os: permite acceder a funcionalidades dependientes del Sistema Operativo. Sobre todo, aquellas que nos refieren información sobre el entorno del mismo y nos permiten manipular la estructura de directorios.

```
>>> os.system("clear")
```

Módulos de Python

The Python Standard Library — Python 3.10.0 documentation

PROGRAMACIÓN ESTRUCTURADA

Introducción

- La **programación estructurada** es un paradigma de programación orientado a mejorar la claridad, calidad y tiempo de desarrollo de un programa de ordenador, utilizando únicamente subrutinas (funciones o procedimientos) y tres estructuras: secuencia, alternativas y repetitivas.

- La **programación modular** es un paradigma de programación que consiste en dividir un programa en módulos o subprogramas con el fin de hacerlo más legible y manejable.

- Al aplicar la programación modular, un problema complejo debe ser dividido en varios subproblemas más simples, y estos a su vez en otros subproblemas más simples. Esto debe hacerse hasta obtener subproblemas lo suficientemente simples como para poder ser resueltos fácilmente con algún lenguaje de programación (divide y vencerás).

- La programación estructural y modular se lleva a cabo en python3 con la definición de funciones.

Definición de funciones

- Declaración

```
>>> def factorial(n):
...     """Calcula el factorial de un número"""
...     resultado = 1
...     for i in range(1,n+1):
...         resultado*=i
...     return resultado
```

- Uso

```
>>> factorial(5)
120
```

Ámbito de variables – local y global

- El ámbito de una variable es el lugar en el código fuente donde esta variable se puede usar.
- Una variable **local** que se declaran dentro de una función no se pueden utilizar fuera de esa función. Ejemplo:

```
>>> def operar(a,b):
...     suma = a + b
...     resta = a - b
...     print(suma,resta)
...
>>> operar(4,5)
9 -1
>>> resta
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'resta' is not defined
```

- Las variables **globales**, serán visibles en todo el módulo. Se recomienda declararlas en mayúsculas.

```
>>> PI = 3.1415
>>> def area(radio):
...     return PI*radio**2
...
>>> area(2)
12.566
```

Parámetros

- **Parámetros formales.**
  - Son las variables que recibe la función, se crean al definir la función.
  - Su contenido lo recibe al realizar la llamada a la función de los parámetro reales.
  - Los parámetros formales son variables locales dentro de la función.

- **Parámetros reales.**
  - Son la expresiones que se utilizan en la llamada de la función, sus valores se copiarán en los parámetros formales.

Paso de parámetros – por valor / por referencia

- En Python el paso de parámetros es siempre **por referencia**. El lenguaje no trabaja con el concepto de variables sino objetos y referencias.
- Al realizar la asignación a = 1 no se dice que “a contiene el valor 1” sino que “a referencia a 1”. Así, en comparación con otros lenguajes, podría decirse que en Python los parámetros siempre se pasan por referencia.
- Aunque podemos cambiar el parámetro real cuando los objetos pasados son de tipo mutables, no es recomendable hacerlo en Python. En otros lenguajes es necesario porque no tenemos opción de devolver múltiples valores, pero en Python podemos devolver tuplas o lista con la instrucción return.

Paso de parámetros

- Si se pasa un valor de un objeto inmutable, su valor no se podrá cambiar dentro de la función:

```
>>> def f(a):
...     a=5
>>> a=1
>>> f(a)
>>> a
1
```

- Si pasamos un objeto de un tipo mutable, si podremos cambiar su valor:

```
>>> def f(lista):
...   lista.append(5)
...
>>> l = [1,2]
>>> f(l)
>>> l
[1, 2, 5]
```

Devolución de información

- Una función en python puede devolver información utilizando la instrucción return.
- La instrucción return puede devolver cualquier tipo de resultados, por lo tanto es fácil devolver múltiples datos guardados en una lista o en un diccionario.
- Cuando la función llega a la instrucción return devuelve el resultado y termina su ejecución.

Llamadas a una función

- Cuando se llama a una función se tienen que indicar los parámetros reales que se van a pasar.
- La llamada a una función se puede considerar una expresión cuyo valor y tipo es el retornado por la función.
- Si la función no tiene una instrucción return el tipo de la llamada será None.

```
>>> def cuadrado(n):
...   return n*n

>>> a=cuadrado(2)
>>> cuadrado(3)+1
10
>>> cuadrado(cuadrado(4))
256
>>> type(cuadrado(2))
<class 'int'>
```

Funciones recursivas

- Una función **recursiva** es aquella que al ejecutarse hace llamadas a ella misma.
- Tenemos que tener “un caso base” que hace terminar el bucle de llamadas.
- Ejemplo:

```
def factorial(num):
    if num==0 or num==1:
        return 1
    else:
        return num * factorial(num-1)
```

```
>>> factorial(5)
120
```