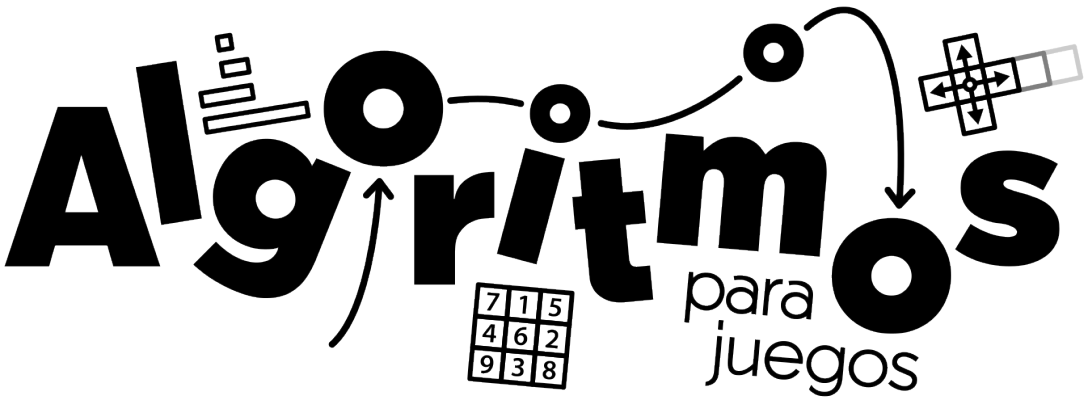



# Tema 0: Introducción a Python

Alejandro Vargas Lugo - Alexvar
algoritmos.neocities.org
Ver. 1.0.3 - 2025.02.07

CC BY NC SA

# Índice



# Prefacio

**Algoritmos para juegos,** parte del Grado en Diseño y Desarrollo de Videojuegos, se presenta como una de las asignaturas más desafiantes de tercer año. Probablemente ya te hayan contado cómo funciona, por estudiantes de cursos superiores que ya han pasado por la asignatura. Seguro que has oído historias de terror de ejercicios imposibles, exámenes de mucha presión y "jueces" que no perdonan ni una.

No te voy a engañar — la asignatura es difícil, en gran medida por el tiempo limitado, la gran cantidad de algoritmos a estudiar y la compaginación con otras asignaturas del mismo cuatrimestre *(¿oigo por ahí Entornos Multijugador?)*, pero quiero que este año sea diferente. Por eso he preparado esta selección de apuntes, ejercicios y contenidos multimedia. Mi objetivo es que no solo consigas aprobar la asignatura, sino que además, en la medida de lo posible, sientas que estás aprendiendo algo útil, y disfrutes mientras refuerzas tus habilidades de programación.

Estos documentos de apuntes cubren todos los aspectos teóricos que debes conocer para ser capaz de resolver los ejercicios de la asignatura, pero eso no significa que los debas tratar como libros de texto aburridos. La mejor forma de consumir estos documentos es con una ventana de programación al lado. No dudes en copiar y pegar los extractos de código, ejecutarlos y ver qué hacen. Anímate a cambiar variables, añadir elementos o borrar funciones enteras, aquí no hay penalización por probar cosas alocadas. Lo peor que te puede aparecer es un error, e incluso eso tiene arreglo.

Esto es Algoritmos para juegos. Mucha suerte y ánimo.

Alejandro Vargas

# Te damos la bienvenida a Python

## ¿Qué es Python?

Python es un **lenguaje de alto nivel** de propósito general, con un enfoque en la **legibilidad y simplicidad de código,** y versatilidad a la hora de realizar programas con él, siendo en la actualidad uno de los lenguajes **más extendidos y demandados** en la industria del software, usado tanto para sencillos scripts de automatización de tareas como grandes proyectos enfocados en Machine Learning.

Es por estos motivos, entre muchos otros, por lo que utilizamos Python en esta asignatura. Su sintaxis y sistema de Estructuras de Datos la hacen ideal para plantear los ejercicios que dentro de no mucho tiempo estarás realizando desde tu ordenador.

## ¿Por qué *no* Java? ¿O C++?

Lo más probable es que ya hayas adquirido práctica con **otros lenguajes de alto nivel** llegados a este punto (Java, C++, C#…), y posiblemente estés pensando…

### ¿Otro más? ¿Por queeeé? ¿No puedo usar Java? :(

Lo cierto es que es comprensible sentir rechazo a tener que aprender de cero otro lenguaje, y más todavía si es para una asignatura que se presenta como una de las más desafiantes del segundo cuatrimestre de tercero o cuarto año. Pero no debes preocuparte. Yo también pasé por esa fase, y créeme, vas a agradecer dar esta asignatura en Python (y si no me crees, pregúntale a la gente que cursó la asignatura antes de 2020, cuando sí se hacía en Java, a ver que te dicen).

Sin entrar en detalles (esos ya los veremos más adelante en este documento), veamos una de las muchas magias de Python. En Java, para imprimir una lista de números del 0 al 5 haríamos algo así:

```
public class Main {
    public static void main(String[] args) {
        for (int i = 0; i < 6; i++) {
            System.out.println(i);
        }
    }
}
```

Fíjate primero en todo el overhead que hemos tenido que crear (declarar clase y método) antes de poder hacer nada, debido a la estructura orientada a objetos de Java. Seguro que el “public static void main” lo tienes grabado a fuego a estas alturas. ¿Y ese print? ¡No me digas que el “system out println loquesea” no te suena ridículo!

Ahora veamos el mismo procedimiento con Python:

```
for i in range(6):
    print(i)
```

Ya está. Sin overhead de métodos y clases y sin corchetes por todos lados. **Python te permite concentrarte en los conceptos de algoritmos sin distraerte con detalles sintácticos.** Esto es muy importante a la hora de estar resolviendo ejercicios de dificultad alta, o en medio de un examen donde los nervios pueden jugar en tu contra.

## Nuestras herramientas de programación

Como buenos profesionales que somos, vamos a utilizar **PyCharm**, uno de los IDEs de Python comerciales más sofisticados, y que además cuenta con una versión Community, completamente gratuita y sin registro, y que cubrirá de sobra todas nuestras necesidades durante el desarrollo de esta asignatura. Puedes obtenerlo desde el sitio oficial de JetBrains, en

https://www.jetbrains.com/es-es/pycharm/download/

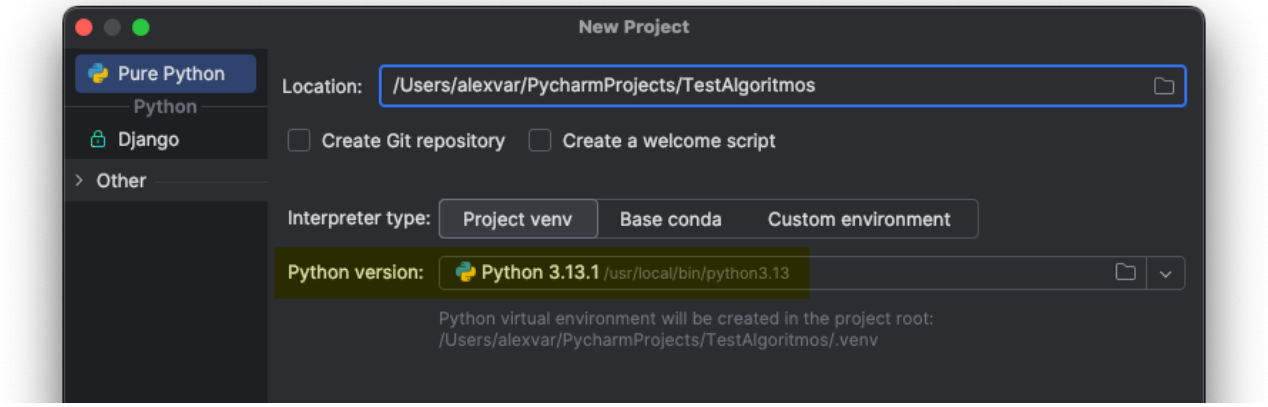
Y desplazándose hasta la parte inferior, donde se encuentra el descargador para la versión Community.


Hay una versión disponible para Windows, otra para distribuciones Linux y otra para macOS tanto en Intel como en ARM (Apple Silicon), por lo que si eres un friki como yo con un MacBook M1, no tendrás problema.

Además de esta IDE, también deberás instalar la última versión de Python desde el sitio oficial, ubicado en:

https://www.python.org/downloads/

Y seleccionando la descarga correspondiente a tu sistema operativo. Con Python instalado, podrás seleccionar el mismo como intérprete de tus proyectos de PyCharm para poder ejecutarlos y depurarlos sin problema.




Al crear un proyecto nuevo, asegúrate de seleccionar "**Project venv**" como tipo de intérprete, y en el desplegable de **Versión de Python**, la que has instalado en el paso anterior.


En la captura de la parte superior puedes ver la interfaz de usuario de PyCharm y dónde están ubicados los elementos principales y que más utilizarás. Si has usado con anterioridad alguna otra IDE de JetBrains, como IntelliJ IDEA, está interfaz te resultará familiar.

Con esto, ya podemos comenzar nuestra aventura en la programación en Python.

# Sintaxis y consideraciones generales

Python difiere de la mayoría de lenguajes de alto nivel en que su código va estructurado mediante **indentaciones,** en lugar de {llaves}. Esto no es solo una cuestión de estilo o legibilidad; es un **requisito** del lenguaje:

```
i = 8
while i <= 10:
    n = 5
    print("Debes respetar las indentaciones")
    if n == 5:
        print("O tendrás errores de sintaxis por todas partes")
        print("¿Lo has entendido?")
```

Además, si te fijas, tampoco se utilizan **puntos y coma al final de cada línea**, así como se **omiten los paréntesis** en las definiciones de los bucles y condicionales (se pueden poner, pero son redundantes) y se usan **dos puntos** para abrir los mismos.

Otro aspecto importante de Python es que usa **tipado dinámico**; es decir, el tipo de una variable se determina y modifica **automáticamente en tiempo de ejecución**, según el valor asignado a esa variable. No hay que definirlo manualmente como ocurre con otros lenguajes (short, long, int, float, char…)

Por otra parte, los nombres de las variables o funciones deben cumplir una serie de requisitos típicos: solo deben usarse caracteres alfanuméricos y el guion bajo “_”, no pueden empezar por número ni usar una palabra reservada. Además, mucho ojo porque Python distingue entre mayúsculas y minúsculas…

```
print("por lo que si pretendes declarar una variable así:")
MiVaRiAbLe = 25
print("y luego acceder a ella de esta manera")
if mivariable == 25:
    print("¡lo llevas claro!")
```

Para crear **comentarios**, se usa una **almohadilla** (o numeral) al principio de la línea,

```
# De esta manera.
```

En Python no hay una forma oficial de hacer comentarios multilínea, pero se puede usar texto encerrado por tres comillas (""") como alternativa (aunque esto, en realidad, lo que hace es definir un string multilínea que Python ignora. Es preferible no usarlas)

## Operadores y expresiones

Los operadores y expresiones usadas a la hora de evaluar valores son los mismos que en la mayoría de lenguajes de alto nivel (ver tabla inferior). Sin embargo, una diferencia a destacar está en los operadores de **negación, conjunción y disyunción**: mientras que en Java se usan `!` `&&` o `||` respectivamente, en Python se usan las palabras **not**, **and** y **or**.

| Nombre | Símbolos | Paridad | Precedencia |
|---|---|---|---|
| Exponente | `**` | Binario | 1 |
| Signo | `+, -` | Unario | 2 |
| Multiplicación/División | `*, /` | Binario | 3 |
| Módulo (división entera) | `//, %` | Binario | 3 |
| Suma/Resta | `+, -` | Binario | 4 |
| Distinto/Igual que | `!=, ==` | Binario | 5 |
| Menor/Mayor (o igual) | `<, <=, >, >=` | Binario | 5 |
| Negación | `not` | Unario | 6 |
| Conjunción | `and` | Binario | 7 |
| Disyunción | `or` | Binario | 8 |

## Tipos de datos y asignación de variables

Como estamos tratando con un lenguaje con tipado dinámico, tendremos que prestar atención a los tipos de datos que manejamos, ya que, aunque Python nos facilita las cosas asignando tipos de datos a variables y reconvirtiéndolos de forma automática y transparente al usuario, en ocasiones será importante convertir de forma explícita a otro tipo de dato para que dichas variables puedan ser procesadas por una función o método sin errores. En cuanto a números, solo distinguimos entre **enteros (int)** y **reales (float).** En cuanto a otros tipos de datos, tenemos **booleanos (bool)** y **cadenas de caracteres (str)**.

Para conocer el tipo de dato de una variable, basta con usar la función **type(variable)**. Esta función puede usarse con cualquier variable conteniendo una estructura de datos para conocer cuál está usando. Estas estructuras las veremos un poco más adelante.

Veamos un ejemplo de cómo Python maneja de forma automática el tipo de dato de una variable con un par de ejemplos…

| Código | Salida |
|---|---|
| a1 = 54<br>print(type(a1))<br>a1 = 26.73<br>print(type(a1))<br>a1 = "hola"<br>print(type(a1)) | <class 'int'><br><class 'float'><br><class 'str'> |
| a1 = 54<br>a2 = 23.76<br>print(type(a1))<br>a1 += a2<br>print(type(a1)) | <class 'int'><br><class 'float> |

Como podemos observar en el segundo ejemplo, si realizamos una operación sobre un entero que lo lleve al terreno decimal, también cambiará su tipo de dato.

Para realizar una conversión explícita de tipo de dato se hacen uso de las funciones `int()`, `float()`, `bool()` y `str()`, dependiendo de a qué tipo de dato queremos convertir la variable:

```
i = 8
print(type(i))

j = float(i)
k = str(i)
l = bool(i)

print(type(j), type(k), type(l))
print(j, k, l)
```

```
<class 'int'>
<class 'float'> <class 'str'> <class 'bool'>
8.0 8 True
```

Al convertir un **número a booleano**, este será **False** solo si es igual a **cero**, y en el caso de las cadenas de texto, solo si la misma **está vacía**. Intentar convertir un string no numérico (“hola”, “veintinueve”, “dQw4w9WgXcQ”…) a un tipo de dato numérico resultará en un error.

Por último, podemos asignar valores a **múltiples variables** al mismo tiempo mediante una lista u otro elemento iterativo (más adelante veremos con detalle las listas, no te preocupes)

```
a, b, c = [5, 8, 7]  # a contiene 5, b contiene 8 y c contiene 7
```

## Módulos

Aunque en esta asignatura no haremos uso extensivo de módulos hasta temas posteriores, es importante conocerlos desde ahora y saber cómo importarlos y trabajar con ellos.

Aunque Python lleva incorporadas muchas de las Estructuras de Datos que usaremos en los ejercicios, algunas como las listas “deque” (double-ended queue, lista con doble terminación; no te preocupes todavía por ellas) forman parte de un módulo externo, en este caso llamado **collections**. Para importarlo, basta con añadir

```
from collections import deque
```

al principio del fichero. Una característica interesante de los módulos es que puedes usar un **alias** para llamar a la función con un nombre distinto al original, por ejemplo, en este caso, podrías usar

```
from collections import deque as dq
```

Y de esta manera, podrías usar `dq` para declarar una lista con doble terminación, en vez de su nombre completo:

```
# De este modo, la declaración de la lista quedaría:
milista = dq()
# en vez de:
otralista = deque()
```

# Condicionales y bucles

## Condiciones: if, elif, else

La estructura de las sentencias condicionales **if** es parecida al de otros lenguajes. Recuerda, como ya dijimos al principio del tema, que Python se estructura con indentaciones en vez de llaves o corchetes.

```
num = 7
correcto = True
if num < 10 and correcto:
    print("Aquí va el contenido del if. No olvides respetar la indentación")
```

Si queremos evaluar otros casos cuando no se cumple la condición del if, usamos **else** y **elif** (else if):

```python
nota = 8.75
if nota < 5:
    print("suspenso")
elif nota < 7:
    print("aprobado")
elif nota < 9:
    print("notable")
else:
    print("sobresaliente")
```

En Python no existen los switch-case; en su lugar tendremos que usar **elif encadenados**.

## Bucles while

Los bucles **while** se ejecutan de forma repetida mientras su condición sea verdadera. Su estructura es prácticamente idéntica al de los if:

```python
i = 0
while i < 10:
    print("Mientras i sea menor que 10, me ejecutaré")
    i += 1
```

En Python no existen los bucles do-while.

## Bucles for. Función range()

Los bucles for en Python funcionan de forma bastante distinta a otros lenguajes de alto nivel. Mientras que en Java o C un bucle for típico tiene la siguiente estructura:

```java
// for (valor inicial; condición/valor final; incremento)
// por ejemplo:

for (int i = 0; i < 10; i++) {
    System.out.println(i);      // Java
    printf(i);                  // C
}
```

En Python sigue esta otra estructura:

```python
# for [valor/elemento] in [rango/iterable]:
# por ejemplo:

for i in range(10):
    print(i)
```

Para iterar en un rango numérico se usa la función **range()**, que admite los **mismos tres argumentos** usados en los bucles for en Java o C, pero puede recibir menos si únicamente es necesario definir el **valor inicial y final** (2 args) o solo el **valor final** (1 arg):

| Java/C | Python |
|---|---|
| `for (int i = 0; i < 12; i++) {` | `for i in range(12):` |
| `for (int i = 4; i < 17; i++) {` | `for i in range(4, 17):` |
| `for (int i = 6; i < 28; i += 3) {` | `for i in range(6, 28, 3):` |

Pero esto es solo una pequeña muestra del poder que tienen los bucles for en Python, y es que, en lugar de un rango numérico, podemos definir **cualquier estructura iterable**:

```
lista = [2, 3, 5, 7, 11, 13, 17, 19]
for elemento in lista:
    print(elemento)
```

Este programa iterará por cada uno de los elementos de la lista y, en este caso, irá imprimiendo cada elemento en una línea distinta.

La gran mayoría de estructuras de datos son iterables, incluyendo listas, tuplas, conjuntos, diccionarios, e incluso cadenas de texto:

```
ciudad = "Santander"
for letra in ciudad:
    print(letra)
```

Este programa iterará por cada una de las letras de la cadena de texto “Santander”, por orden, y las irá imprimiendo.

Por último, si solo necesitamos ejecutar un fragmento de código un número determinado de veces, y **no vamos a hacer referencia a un índice**, lo podemos **omitir** sustituyéndolo por un guion bajo:

```
for _ in range(5):
    print("Me voy a ejecutar cinco veces”)
```

# Estructuras de datos básicas

## Listas

La **lista** es la estructura de datos lineal más importante de Python. Permite guardar una serie de elementos **de forma ordenada e indexada**. Estos elementos pueden ser **cualquier tipo de dato**, o incluso otras estructuras de datos. Los elementos **pueden repetirse**. Como es común en programación, **los índices empiezan por cero**.

Para inicializar una lista usamos los **[corchetes]**:

```
lista = [4, 7, 2.91, "datos", "algoritmo", True]
```

Para acceder a un elemento de la lista, solo hay que indicar su índice entre corchetes:

```
mielem = lista[4]  # Ahora mielem contiene el string “algoritmo”
```

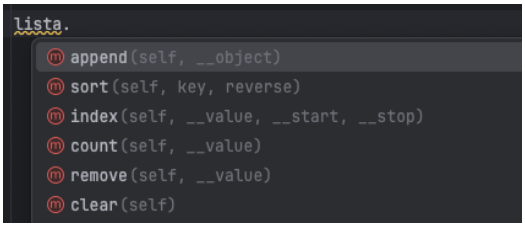
De forma similar, para modificar un elemento, solo hay que asignarle un valor distinto:

```
lista[2] = 6.91  # Ahora el tercer elemento contiene el número 6.91
```

También podemos crear **listas vacías**:

```
listavacia = []
```

Y posteriormente añadir elementos a la lista usando uno de los muchos **métodos** disponibles. Hablando de ellos, veamos los más importantes a continuación. Es conveniente mencionar que **no es necesario** aprenderse todos los métodos de memoria, ya que podremos consultarlos desde PyCharm, incluso durante el examen.




- `append(elem)` - agrega el elemento especificado al final de la lista.
- `remove(elem)` - elimina el elemento especificado. Si dicho elemento no está en la lista, se produce un error.
- `index(elem)` - devuelve el índice del elemento especificado. Si dicho elemento está repetido, devuelve el índice más bajo. Si no está en la lista, se produce un error.

- `sort(arg)` - ordena la lista por orden ascendente. Si se especifica como argumento **reverse=True**, se realiza por orden descendente. Si la lista contiene números y cadenas de texto se produce un error.
- `count(elem)` - devuelve el número de ocurrencias del elemento especificado en la lista.
- `pop(index)` - elimina y devuelve el elemento situado en el índice especificado. Si el índice se encuentra fuera del rango de la lista, se produce un error.

Antes de terminar, un aspecto importante: **¡las listas son punteros!** Esto significa que, si creamos una nueva variable que apunte a una lista, no se creará una copia, sino que ambas variables apuntarán a la misma lista. El resto de estructuras de datos que veremos en esta sección también poseen esta característica.

## Matrices con listas

La forma más cómoda de trabajar con matrices de datos en Python es manejando una **lista de listas**.

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

También se puede definir en una única línea, pero separado de esta manera resulta más fácil de visualizar.

Para acceder a un elemento de la matriz, se usan dos pares de corchetes. ¡Ojo! Ten en cuenta que el primer número indica la fila y el segundo la columna. Y no olvides que los índices empiezan en cero:

```
cifra = matriz[3][1]  # Ahora cifra contiene el número 2
```

## Tuplas

Las **tuplas** son sencillamente listas que **no son mutables**, es decir, una vez que se han inicializado, no se puede modificar de ninguna forma, ni añadiendo, quitando o modificando sus elementos. Por su naturaleza de solo lectura, son más rápidas y eficientes en uso de memoria que las listas.

Para inicializar una tupla usamos los **(paréntesis)**:

```
tupla = (3, "impresora", 7.65, False, "Opel Corsa”)
```

Las tuplas solo tienen dos métodos principales: **index** y **count**.

## Conjuntos

El **conjunto** es una estructura de dato **no lineal** en la que sus elementos **no están ordenados, ni indexados, ni pueden ser modificados** (pero sí es posible añadir y quitar elementos). Si se añade uno o más elementos que ya están en el conjunto, **se descartan**.

Para inicializar un conjunto usamos las **{llaves}**:

```
conjunto = {"alex", "bea", "celia", "dani", "eli", "fede", “gema"}
```

El orden en el que coloquemos los elementos no importa. De hecho, si intentamos iterar sobre el conjunto e imprimir los elementos…

```
for elemento in conjunto:
    print(elemento)
```

El orden en el que se imprimirán será aleatorio. En el caso de los conjuntos de strings, el orden será aleatorio en cada ejecución del programa, ya que sus valores hash se aleatorizan en cada ejecución por seguridad. (No tienes que saber esto último)

Los métodos disponibles son parecidos a los de las listas, aunque algunos con nombres diferentes:

- `add(elem)` - agrega el elemento especificado al conjunto.
- `remove(elem)` - elimina el elemento especificado. Si dicho elemento no está en el conjunto, se produce un error.
- `discard(elem)` - igual que remove, pero **no produce un error** si el elemento no está en el conjunto.
- `pop()` - elimina y devuelve un elemento aleatorio del conjunto. Si el mismo está vacío, se produce un error.

Un aspecto interesante de los conjuntos es que cuentan con **funciones de operaciones booleanas**:

- `union(conj)` - devuelve la unión con el conjunto especificado.
- `difference(conj)` - devuelve la diferencia (resta) con el conjunto especificado.
- `intersection(conj)` - devuelve la intersección con el conjunto especificado.

Veamos un ejemplo:

```
con1 = {"alex", "bea", "celia", "dani", "eli", "fede", "gema"}
con2 = {"eli", "fede", "gema", "hector", "isma", "jade", “kim"}

con3 = con1.union(con2)        # con3 contiene todos los nombres sin repetir.
con3 = con1.difference(con2)   # con3 contiene alex, bea, celia y dani.
con3 = con1.intersection(con2) # con3 contiene eli, fede, gema.
```

## Diccionarios

Los diccionarios son estructuras lineales consistentes en **pares de datos clave-valor**. Los datos están **ordenados e indexados**, pero **no puede haber claves repetidas**. Pueden ser útiles al trabajar con datos pareados.

Para inicializar un conjunto también usamos **{llaves},** pero la estructura interna es distinta a la de los conjuntos, por supuesto:

```
info = {
    "Nombre": "Juan",
    "Apellido": "Pérez",
    "Fecha": 2004,
    "Altura": 182
}
```

Para acceder a un valor, podemos hacerlo a partir de su clave:

```
fecha_nac = info[“Fecha"]  # Ahora fecha_nac contiene el número 2004
```

Para añadir un nuevo par clave-valor no se usa un método, sino que directamente se crea una nueva clave a la que se le asocia un valor:

```
info["DNI"] = “10485760B"
```

Para modificar un elemento, también basta con referenciar su clave:

```
info["Fecha"] = 2002  # Ahora la clave "Fecha" contiene 2002, y no 2004
```

Los diccionarios también cuentan con múltiples métodos para manejar sus datos de forma más avanzada.

# Todo sobre entradas y salidas

## Salida de texto: función *print()*

Como ya habrás podido observar a lo largo de los ejemplos que se han ido utilizando, la función para mostrar texto por la salida estándar (consola) es print().

```
print("Todo lo que pongas entre comillas se mostrará tal cual")
```

Pero si quieres imprimir el valor de una variable,

```
# deberás omitir las comillas, como en el siguiente ejemplo:
valor = 32
frase = "Jovencillo emponzoñado de whisky"
print(valor)
print(frase)
```

La salida por consola será el número 32, seguido de la frase “Jovencillo emponzoñado de whisky” en una nueva línea. De hecho, al igual que en otros lenguajes, puedes evaluar una expresión o una función dentro de un print:

```
var1 = 32
var2 = 24
var3 = 18
var4 = 24
print(var1 + var2 + var3)
print(var2 == var4)

# La primera línea mostrará el número 74, mientras que la siguiente
# mostrará True
```

Una de las magias de la función print() en Python es que puedes **imprimir cualquier cosa directamente**. Ya otra cosa son las suposiciones que haga Python sobre el tipo o estructura de datos que le estés pasando:

```
milista = [1, 2, 4, 8, 16, 32, 64, 128]
print(milista)

# Mostrará, literalmente por pantalla: [1, 2, 4, 8, 16, 32, 64, 128]
```

Si quisieras mostrarlo con otro formato, por ejemplo, sin corchetes ni comas, entonces ya tendrías que recurrir a un **bucle for**.

## Concatenando elementos

Existen varias formas de concatenar elementos en un **print()**. La más directa consiste en usar el símbolo de suma. Sin embargo, a diferencia de Java, todos los elementos a concatenar deben ser de **tipo string (str)** al usar este sistema. Se debe usar, pues, la función **str()** para convertir los números o booleanos a una cadena de caracteres.

```
altura = 114
print("la altura de las Torres KIO es de " + str(altura) + " metros”)
```

```
la altura de las Torres KIO es de 114 metros
```

Este sistema **no es ideal** cuando tenemos muchos números y cadenas de caracteres que juntar, así que la mejor opción consiste en usar los **print formateados**:

```
version = 7
fecha = 1997
ventas = 14.4
print(f"Final Fantasy {version} salió en {fecha} y vendió {ventas} millones
de unidades”)
```

```
Final Fantasy 7 salió en 1997 y vendió 14.4 millones de unidades
```

Como alternativa a la primera opción también se pueden usar comas en vez de símbolos de suma, esto tiene la ventaja de no requerir convertir los elementos a cadenas de texto, pero con la “desventaja” de que añaden automáticamente un espacio entre elementos:

```
version = 64
fecha = 1996
print("Super Mario", version, "salió en", fecha, "para la Nintendo", version)
```

```
Super Mario 64 salió en 1996 para la Nintendo 64
```

Así que si por cualquier motivo quisieses concatenar sílabas para formar una palabra…

```
N1 = 348
S1 = "cho"
S2 = "co"
S3 = "la"
S4 = "te"
print("Me han regalado", N1, "tabletas de", S1, S2, S3, S4)
# Te llevarías un chasco, porque te regalarían cho co la te, y no chocolate.
```

```
Me han regalado 348 tabletas de cho co la te
```

## Entrada de texto I: input()

Esta es quizá una de las partes **más importantes** de este tema introductorio, y es que en los ejercicios de algoritmos **siempre** deberás procesar datos de entrada y hacerlo correctamente. El no hacer esto puede llevarte a muchos quebraderos de cabeza con errores muy tontos, así que presta atención:

La función para recibir un texto de entrada es `input()`. Cuando se ejecuta, el programa se detiene esperando una línea de texto de la consola. Dicha entrada **siempre se procesa como un string**, independientemente de su contenido. Por ejemplo, si nuestro programa contiene la siguiente línea:

```
nombre = input()
```

E introducimos el siguiente texto y le damos a Intro:

```
Juan Pérez
```

Dicho string se guardará en la variable "nombre" y la ejecución del programa continuará. Recuerda que, independientemente de lo que hubiéramos escrito, "nombre" **siempre será de tipo string**, incluso si en vez de Juan Pérez, hubiésemos puesto algo como:

```
16777216
```

Esto es **MUY IMPORTANTE** porque si queremos trabajar con los valores de entrada como números, será obligatorio convertirlos a enteros (int) o reales (float). Esto se puede hacer directamente en la llamada a la función input:

```
num_telf = int(input())
```

De esta forma, nos aseguraremos de que el texto de entrada sea convertido a un entero antes de ser asignado a nuestra variable.

## Entrada de texto II: split(), map() y list()

En múltiples ocasiones tendremos que procesar una única línea de texto donde hay muchos valores que tendremos que tratar como variables distintas. Esto lo podemos conseguir utilizando la función **split()**. Por ejemplo, imaginemos que llegamos a una función **input()** y recibimos la siguiente cadena de caracteres:

```
24 91 75 37 86
```

Si seguimos el procedimiento anterior, se guardará toda la línea en una única variable, cosa que no queremos, por lo que si debemos guardar cada número como su propia variable, 5 en total, una solución sería:

```
a, b, c, d, e = input().split()
```

**split()** toma la línea recibida como entrada por **input()**, divide cada elemento separado por espacios y devuelve una lista. A efectos internos esto es lo que estaría viendo Python:

```
a, b, c, d, e = ["24", "91", "75", "37", “86"]
```

Que es justo lo que queríamos. Sin embargo, hay un problema importante: estas variables son **cadenas de caracteres**, y nosotros necesitamos enteros. Podríamos cambiar el tipo de dato uno a uno, pero eso sería tedioso e innecesario, más todavía con listas de 10, 20 o más elementos.

¿La solución? La función map():

```
a, b, c, d, e = map(int, input().split())
```

Que hace tal y lo que dice, “mapear” la función pasada como primer argumento a los elementos del segundo argumento. En este caso, aplicar la función int a cada uno de los elementos recogidos por el input, convirtiéndolos de cadenas de caracteres a enteros. Internamente, esto es aproximadamente lo que está sucediendo:

```
a, b, c, d, e = [int("24"), int("91"), int("75"), int("37"), int(“86")]
```

Si nuestros valores fuesen números reales en vez de enteros, sería tan sencillo como cambiar el primer argumento de int a float en la función map().

Ahora bien, ¿y si quisiéramos guardar nuestros números en una lista directamente? Aquí es donde la función list() entra en acción:

```
lista = list(map(int, input().split()))
```

Ahora los números que antes estaban almacenados en cinco variables distintas, lo están en una única lista más conveniente de manejar.

## Interpretando ejercicios

Si te fijas en cualquier ejercicio de esta asignatura, verás una tabla de entrada y salida como la siguiente:

| Entrada | Salida |
|---|---|
| 13<br>21<br>17 | 51 |

En este caso recibimos **tres líneas de entrada** con un número en cada uno. La salida consiste en la suma de esos tres números. En cualquier caso será necesario llamar a la función input() **tantas veces como líneas de entrada** haya en el ejercicio; la forma más práctica de conseguir esto es mediante el uso de un **bucle for** y una **lista vacía**:

```
lista_nums = []
for _ in range(3):
    a = int(input())
    lista_nums.append(a)
```

De hecho, las dos líneas dentro del bucle for se pueden condensar en una, así:

```
for _ in range(3):
    lista_nums.append(int(input()))
```

Ambas soluciones son perfectamente válidas. Te encontrarás con muchos casos así donde puedes (o no) condensar líneas de código. Si te sientes más cómodo/a trabajando con código más legible, puedes optar por la primera opción.

A continuación, a sumar los números:

```
suma = 0
for num in lista_nums:
    suma += num
```

Y por último, a mostrarlo por pantalla:

```
print(suma)
```

He mostrado el proceso paso a paso para hacerlo más fácil de seguir y para ejemplificar algunos conceptos, pero cuando adquieras práctica, seguro que lo resolverás así:

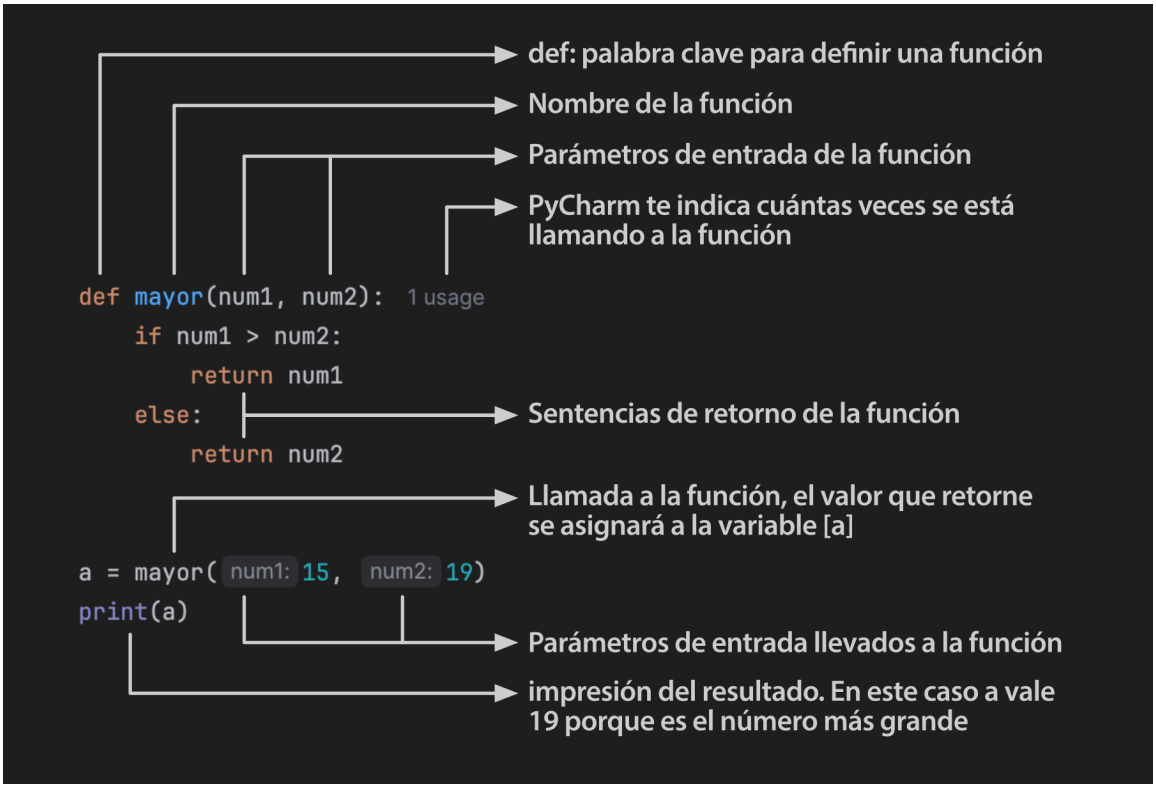
```
suma = 0
for _ in range(3):
    suma += int(input())
print(suma)
```

¡Ni siquiera nos hacía falta guardar los elementos en una lista! Como puedes ver, incluso en los ejercicios más sencillos se pueden encontrar optimizaciones muy interesantes. No te sientas mal si no lo hubieses visto, uno de los objetivos de esta asignatura es el de reforzar lo que yo llamo el **"pensamiento algorítmico"**. Cuando comiences a resolver los ejercicios que tenemos preparados para ti, empezarás a ganar destreza a la hora de encontrar estos atajos y mejoras, e irás adquiriendo una versatilidad a la hora de programar que no solo te ayudará en esta asignatura, sino que además te será crucial en el mundo profesional a la hora de implementar funciones en grandes proyectos informáticos.

# Funciones

A lo largo de este tema hemos visto muchas de las funciones integradas de Python, como int(), float(), str(), type(), range(), input(), map(), split()… Pero tú también puedes hacer tus propias funciones de manera sencilla.

Una **función** es un **bloque de código que realiza una tarea específica** y que puede ser llamado desde cualquier parte de tu programa. Una función puede recibir argumentos de entrada, devolver (retornar) uno o varios valores, o ambos. Las funciones en Python se definen con la palabra clave **def.** Veamos la estructura general de una función:



## Alcance de las variables

El alcance *(scope)* de una variable indica las partes de nuestro fichero Python desde donde se puede acceder la misma. Una variable solo se encuentra disponible desde la región donde se ha declarado.

Por ejemplo, una variable creada dentro de una función tiene alcance dentro de esa función, y por lo tanto **no podrá ser usada fuera de ella**. Se dice que esa variable tiene **alcance local**. Por contra, una variable que se haya declarado en la raíz de nuestro fichero Python (fuera de cualquier función, bucle o sentencia condicional) podrá ser accedida desde cualquier parte, y se dice que tiene **alcance global**.

Pero, ¿qué pasa si declaramos dos variables con el mismo nombre, una con alcance global y otra con alcance local, en una función?

| Código | Salida |
|---|---|
| `def variables():`<br>`  i = 64`<br>`  return i`<br><br>`i = 32`<br>`print(variables())`<br>`print(i)` | `64`<br>`32` |

Como podemos observar, la variable con alcance local **sobreescribe (ignora)** el valor de la variable con alcance global dentro de la función, lo que significa que podemos tratarlos como **dos variables independientes**. Sin embargo, es recomendable mantener en lo posible **nombres diferentes de variables** para **evitar confusiones**, y de hecho PyCharm también avisa de ello, y recomienda cambiar el nombre de la variable en dichos casos.