¡Hola, Python!

Diego Morales

2024-07-03

Tabla de contenidos

Prefacio

Bienvenid@ a mi curso de introducción a programación en Python.

He creado este libro con el objetivo de compartir mis conocimientos con todo el mundo de la manera más efectiva posible.

And what if instead of it being in the right hands, it was in everyone's hands? -Steve Jobs [Film] (2015)

1 Configuración

En este capítulo realizaremos la configuración de Python para crear códigos. Iniciaremos explorando algunas de las opciones disponibles y luego mostraremos las instrucciones para instalar los programas en nuestra computadora, o bien, para acceder al servicio en la nube.

1.1 Instalación de Python

1.1.1 Windows

1. Ve al sitio oficial de Python.
2. Haz clic en "Downloads" en la parte superior.
3. Selecciona "Download Python [versión actual]" para descargar el instalador de Windows.
4. Abre el archivo descargado (generalmente llamado python-[versión]-amd64.exe o similar).
5. Marca la casilla "Add Python to PATH" al principio del instalador.
6. Haz clic en "Install Now".
7. Una vez finalizada la instalación, Python y pip (el gestor de paquetes de Python) estarán disponibles en tu sistema.

1.1.2 macOS

1. Ve al sitio oficial de Python.
2. Haz clic en "Downloads" en la parte superior.
3. Selecciona "Download Python [versión actual]" para descargar el instalador de macOS.
4. Abre el archivo descargado (python-[versión]-macosx10.[n].pkg o similar).
5. Sigue las instrucciones del instalador y acepta las configuraciones predeterminadas.
6. El instalador configurará Python y pip en tu sistema.

1.2 Entornos de Desarrollo

Un entorno de desarrollo es una aplicación o conjunto de aplicaciones que facilita la escritura, prueba y depuración del código. Dependiendo de nuestras necesidades y nivel de experiencia, podemos optar por diferentes tipos de entornos. A continuación, se describen brevemente algunos de los entornos de desarrollo más comunes y útiles para programar en Python

1.3 Editores de Texto

Un editor de texto es una herramienta básica que permite escribir y editar código. Ejemplos populares incluyen Notepad++ (Windows), Sublime Text y Atom. Aunque carecen de funcionalidades avanzadas como la depuración, son ligeros y rápidos, ofreciendo características como el resaltado de sintaxis y el autocompletado.

1.4 IDLE de Python

IDLE es el entorno de desarrollo oficial que viene con la instalación de Python. Es ideal para principiantes debido a su simplicidad. Incluye un editor de texto con resaltado de sintaxis, una consola interactiva y herramientas básicas de depuración. Es una opción ligera y fácil de usar para empezar a programar en Python sin necesidad de instalar software adicional.

1.5 Anaconda: Jupyter Notebook y Spyder

Anaconda es una distribución de Python que incluye una gran cantidad de paquetes científicos y herramientas de desarrollo. Jupyter Notebook es una aplicación web que permite crear y compartir documentos que contienen código, texto, y visualizaciones, siendo muy popular en el ámbito científico y educativo. Spyder es un entorno de desarrollo interactivo (IDE) que se asemeja a MATLAB, con un editor avanzado, herramientas de depuración y un explorador de variables.

1.6 Visual Studio Code

Visual Studio Code (VS Code) es un editor de código fuente desarrollado por Microsoft. Es altamente configurable y soporta una gran variedad de lenguajes de programación, incluyendo Python. Ofrece características avanzadas como depuración integrada, control de versiones, extensiones para el desarrollo en Python y una terminal incorporada. Es una de las herramientas más completas y populares entre los programadores.

1.7 Google Colab

Google Colab (Colaboratory) es un servicio basado en la nube que permite ejecutar cuadernos Jupyter en la infraestructura de Google. Es especialmente útil para el aprendizaje automático y análisis de datos, ya que proporciona acceso a recursos de computación potentes de forma

gratuita. Permite colaborar en tiempo real con otros usuarios y no requiere instalación de software en el equipo local.

1.8 PyCharm

PyCharm es un entorno de desarrollo integrado (IDE) específicamente diseñado para Python, desarrollado por JetBrains. Tiene dos versiones: una comunidad gratuita y una versión profesional de pago con más funcionalidades. Ofrece características avanzadas como autocompletado de código, refactorización, depuración integrada, pruebas unitarias y un poderoso editor. Es muy útil para proyectos grandes y complejos.

2 Introducción

En este capítulo creamos nuestro primer código de programación en Python, el cual mostrará un saludo en pantalla al ejecutar el programa.

2.1 Mi primer programa

La función `print` permite *desplegar mensajes* en la pantalla. Escribe el siguiente código teniendo cuidado de escribir correctamente el nombre de la función, luego, dentro de paréntesis deberás colocar tu mensaje utilizando comillas simples, `'`, o dobles, `"`, al inicio y al final.

```
print('¡Hola, Python! 2024')
```

```
¡Hola, Python! 2024
```

Si Python interpretó correctamente tu instrucción, entonces, deberías ver el mensaje anterior en la salida.

> **Nota**
>
> Es una práctica común crear el código para desplegar un mensaje de `Hello, world!` cuando se aprende un lenguaje nuevo de programación para comprobar que todo se encuentra configurado correctamente.

3 Variables

Una variable es un espacio de memoria en el cual podemos almacenar valores.

Por ejemplo, en el siguiente código. Se asigna el valor `4` a una variable llamada `x`.

```
x = 4
```

3.1 Características

Las variables poseen las siguientes características:

1. Puede almacenar distintos tipos de datos.
2. Su valor se puede modificar.
3. Se referencian por un nombre único (idealmente, significativo).

Observaciones sobre nombres de variables:

1. Python diferencia entre mayúsculas y minúsculas.
2. No pueden iniciar con números.
3. No pueden contener espacios.
4. No se pueden utilizar **palabras reservadas** (más adelante aprenderemos qué significan).

```
a = 100
print(a)
```

```
100
```

Si asignamos un nuevo valor a la misma variable, el valor anterior es reemplazado.

```
a = 100
a = 200
print(a)
```

```
200
```

Si se desean crear variables utilizando nombres significativos conformados por varias palabras, se recomienda utilizar la "notación de camello" (*camelCase*):

```
correoElectronico = 'juan_lopez2020@gmail.com'
```

También pueden separarse las palabras utilizando **guion bajo**, sin embargo, este tipo de notación suele utilizarse para funciones.

```
correo_electronico = 'juan_lopez2020@gmail.com'
```

4 Tipos de datos

Existen una gran cantidad de tipos de datos en Python: *Texto, numéricos, secuencia, mapeos, conjuntos, booleanos, binarios.*

4.1 Entero (int)

Dato numérico utilizado para representar números enteros.

```
cantidadAlumnos = 256
print(cantidadAlumnos)
```

```
256
```

4.2 Flotante (float)

Dato numérico utilizado para representar números reales, es decir, números con cifras decimales.

```
alturaMetros = 1.75
print(alturaMetros)
```

```
1.75
```

4.3 Texto (str)

Cadena de caracteres: letras, números y símbolos. El valor **se debe colocar dentro de comillas simples o dobles** para indicar que es un `str`.

```
movie = "The Lord of the Rings: The Return of the King (2003)"
cantidadAlumnos2 = "256"
alturaMetros2 = "1.75"
print(movie)
print(cantidadAlumnos2)
print(alturaMetros2)
```

```
The Lord of the Rings: The Return of the King (2003)
256
1.75
```

> **Advertencia**
>
> Se debe tener especial cuidado al manejar números representados como cadenas de caracteres ya que Python no los interpretará como valores numéricos (`int` o `float`).

4.4 Función type

La función `type()` se puede utilizar para conocer el tipo de dato de una variable.

```
print(type(cantidadAlumnos))
print(type(cantidadAlumnos2))
```

```
<class 'int'>
<class 'str'>
```

4.5 Función input

La función `input()` se utiliza para solicitar al usuario una entrada. El programa se detiene, hasta que el usuario presiona la tecla **Enter** en su teclado (luego de escribir el ingreso). Esta función, retorna una cadena de caracteres (`str`) con el ingreso del usuario.

```
nom = input("¿Cuál es su nombre? ")
```

Por lo tanto, puedes utilizar esta función para asignar valores a variables al iniciar la ejecución de tu programa.

Ahora veamos qué sucede si solicitamos al usuario un número que utilizaremos para realizar una operación más adelante.

```
age = input("Ingrese su edad: ")
```

Esa línea de código sería equivalente a que tomemos el valor devuelto por la función `input` y lo asignemos en la variable `age`. Sería igual al siguiente código.

```
age = '28'
birthYear = 2024 - age
print("Usted nació en el año ", birthYear)
```

Python nos muestra un mensaje de error (TypeError) debido a que la operación de resta (`-`) no está soportada entre variables numéricas (`int` o `float`) y cadenas de caracteres (`str`).

> **!** Importante
>
> Es completamente normal (y esperado) obtener errores al escribir código, incluso para programadores experimentados. Suelen aparecer cuando Python no es capaz de interpretar una instrucción (SintaxError) o porque se está realizando una acción inválida (como la anterior). Lo importante es comprender el error y corregirlo, lo cual será cada vez más sencillo con la práctica.

En este caso, podríamos realizar una conversión de datos para que la variable `age` se interprete como un entero.

```
age = '28'
age = int(age) # conversion a entero (casting)
birthYear = 2024 - age
print("Usted nació en el año ", birthYear)
```

```
Usted nació en el año  1996
```

4.6 Conversión de datos

Para convertir entre distintos tipos de datos, simplemente debemos especificar el tipo de dato y colocar el valor dentro de paréntesis.

1. **int()**
2. **float()**
3. **str()**

Por ejemplo, si queremos convertir una variable **`x = "99"`** a un valor numérico.

```
x = "99"
print(type(x))
x = int(x)
print(type(x))
```

```
<class 'str'>
<class 'int'>
```

5 Operadores Aritméticos

Los operadores ariméticos se utilizan para realizar operaciones matemáticas sencillas sobre valores numéricos:

1. Adición
2. Sustracción
3. Multiplicación
4. División
5. Exponenciación
6. División de piso
7. Módulo

5.1 Adición

La operación de suma entre un valor `a` y un valor `b` se realiza utilizando el operador `+`: $a + b$

> **Tip**
>
> Se pueden sumar múltiples valores en una misma expresión. Por ejemplo: $a + b + c + d$

```
100 + 717.345 + 0
```

```
817.345
```

5.2 Sustracción

La operación de resta entre un valor `a` y un valor `b` se realiza utilizando el operador `-`: $a - b$

> **Precaución**
>
> Nótese que la siguiente operación debería dar un resultado exacto, sin embargo, las computadoras manejan "números de punto flotante" los cuales no poseen una representación binaria de valores limitada y podrían mostrar valores poco precisos.

```
55.54 - 105.54
```

```
-50.00000000000001
```

5.3 Multiplicación

Para multiplicar un número **a** y un número **b**, se utiliza el operador ***.

$a * b$

```
7*2
```

```
14
```

5.4 División

Para dividir un número **a** y un número **b**, se utiliza el operador /.

$\frac{a}{b}$

```
15/2
```

```
7.5
```

5.5 Exponenciación

Para elevar un número **a** al exponente de un número **b**, se utiliza el operador ****.

a^b

```
2**3
```

```
8
```

5.6 Módulo

Para obtener el residuo de la división de un número **a** y un número **b**, se utiliza el operador %.

```
6%6
```

```
0
```

```
7%6
```

```
1
```

5.7 División de piso

Para obtener la división de piso de un número **a** y un número **b**, se utiliza el operador //.

Funciona de forma similar a la división normal, con la diferencia que retorna un número entero e ignora la parte decimal.

```
10//3
```

```
3
```

5.8 Jerarquía de operaciones

Es importante tener en cuenta la jerarquía de operaciones al utilizar varios operadores aritméticos en una misma expresión.

Las operaciones se efectúan en el siguiente orden: 1. Expresiones dentro de paréntesis: () 2. Exponente: ** 3. Multiplicación, División, División entera, Módulo: *, /, //, % 4. Suma, Resta: +, -

Si quisieramos convertir 100°F a °C. Se utilizaría la siguiente expresión:

$(100 - 32) * 5/9$

Siguiendo la jerarquía de operaciones, se realizan de la siguiente manera:

- $(100 - 32) * 5/9$
- $(68) * 5/9$

- 37.78

```
(100 - 32) * 5/9
```

```
37.77777777777778
```

Si no se hubieran colocado los paréntesis, obtendríamos un resultado erróneo: * $100 - 32 * 5/9$ * $100 - 32 * 0.5556$ * $100 - 17.78$ * 82.22

```
100 - 32 * 5/9
```

```
82.22222222222223
```

6 Operadores de asignación

Los operadores de asignación se utilizan para asignar un valor a una variable.

Por ejemplo, si queremos asignar un valor `200.0` a una variable `x` utilizamos el operador `=` de la siguienta manera:

```
x = 200
```

> **Nota**
>
> Nótese que la asignación se realiza de la derecha de la igualdad a la izquierda: $x \leftarrow 200.0$

```
100 + 717.345 + 0
```

```
817.345
```

6.1 Sustracción

La operación de resta entre un valor `a` y un valor `b` se realiza utilizando el operador `-`: $a - b$

> **Precaución**
>
> Nótese que la siguiente operación debería dar un resultado exacto, sin embargo, las computadoras manejan "números de punto flotante" los cuales no poseen una representación binaria de valores limitada y podrían mostrar valores poco precisos.

```
x = 200.0
print(x)
```

```
200.0
```

Los resultados de expresiones se pueden almacenar en variables.

Por ejemplo, si quisieramos sumar el costo de tres productos de una compra se suma el precio de cada uno (`249.99 + 9.99 + 50`) y se almacena el resultado final en una variable (`costoTotal`).

```
costoTotal = 249.99 + 9.99 + 50
print(costoTotal)
```

```
309.98
```

Esto nos permite almacenar resultados que podríamos utilizar posteriormente en el código.

Por ejemplo, asumamos que se le aplica al cliente un descuento del 20% sobre el total de la compra. Ya que tenemos una variable con el costo total (`costoTotal`) solo habría que multiplicar (`*`) por el porcentaje a pagar (`1-descuento`) y almacenar el resultado final en otra variable (`costoFinal`).

```
descuento = 0.20
costoFinal = costoTotal*(1-descuento)
print(costoFinal)
```

```
247.98400000000004
```

Otra ventaja de implementar variables en lugar de colocar valores fijos es que podemos modificar su valor para obtener distintos resultados.

Por ejemplo, si queremos calcular el área de un rectángulo de 20 metros por 40 metros, la fórmula sería:

$rea_{rectngulo} = 20 * 40$

El resultado siempre retornará 800.

$rea_{rectngulo} = base * altura$

Podríamos definir una variable para la base y otra para la altura.

De esta manera, si queremos calcular el área de otro rectángulo, solo debemos cambiar los valores de esas variables.

```
base = 35
altura = 40
areaRectangulo = base*altura
print(areaRectangulo)
```

```
1400
```

6.2 Operadores de asignación y aritméticos

Además del operador de asignación =, existen otros operadores de asignación que realizan operaciones aritméticas y asignan el resultado a la vez.

Si tenemos una variable **x = 3** y queremos aumentar 2 su valor. Utilizando el operador =. Haríamos lo siguiente:

```
x = 3
print(x)
x = x + 2
print(x)
```

```
3
5
```

Si tenemos una variable **cuotasRestantes = 24** y queremos restarle 6. El código sería:

```
cuotasRestantes = 24
print(cuotasRestantes)
cuotasRestantes = cuotasRestantes - 6
cuotasRestantes = cuotasRestantes - 3
print(cuotasRestantes)
```

```
24
15
```

En los ejemplos anteriores, para aumentar o disminuir el valor de una misma variable. Primero, se lee el valor actual de la variable, se realiza la operación y, finalmente, se asigna el resultado a la misma variable.

```
tareas = tareas + 2
```

En estos casos especiales cuando queremos modificar el valor de una variable y almacenar el resultado en la misma variable. Podemos hacer uso de los operadores de aritméticos en cojunto con la asignación. De la siguiente forma:

```
tareas += 2
```

Primero, colocamos la variable que queremos aumentar, luego, el operador aritmético seguido del operador de asignación, y finalmente, colocamos el valor.

Los dos códigos anteriores son equivalentes.

```
horasEstudio = 1
print(horasEstudio)
horasEstudio = horasEstudio + 2
print(horasEstudio)
```

```
1
3
```

```
horasEstudio = 1
print(horasEstudio)
horasEstudio += 2
print(horasEstudio)
```

```
1
3
```

Se puede utilizar cualquier operador aritmético y también podríamos sumar valores de variables en lugar de constantes.

```
calificacion = 90
print("Calificación inicial", calificacion, "pts")

extras = 10
calificacion += extras
print("Calificación con extra", calificacion, "pts")

bono = 1.1
calificacion *= bono
print("La calificación final con el extra y el bono es", calificacion)
```

```
Calificación inicial 90 pts
Calificación con extra 100 pts
La calificación final con el extra y el bono es 110.00000000000001
```

7 Operadores de comparación

Los operadores de comparación se utilizan para comparar dos operandos. El resultado de la operación retornará un valor booleano (`True` o `False`).

Las comparaciones que se pueden realizar son las siguientes:

1. Igualdad: `==`
2. Desigualdad: `!=`
3. Menor que: `<`
4. Mayor que: `>`
5. Menor o igual que: `<=`
6. Mayor o igual que: `>=`

7.1 Igualdad

Si el operando de la **izquierda** es igual al operando de la **derecha**, entonces, el resultado es `True`. De lo contrario, es `False`.

Por ejemplo, podríamos crear las siguientes variables.

```
a = 5
b = 7
c = 5
```

Al evaluar la igualdad entre `a` y `b`, lo escribiríamos de la siguiente manera y obtendríamos un resultado `True`.

```
a == b
```

```
False
```

> **Nota**
>
> Nótese que el resultado es el mismo si hubiéramos escrito los operandos al revés, es decir, `b == a`.

Si ahora evaluamos la igualdad entre `a` y `c`, el resultado será `False`.

```
a == c
```

```
True
```

Finalmente, creamos una variable de tipo `str` con el valor de `'5'` y comparamos si es igual a la variable `a`.

```
d = '5'
d == a
```

```
False
```

A simple vista, los valores de cada variable son idénticos. Sin embargo, Python no considera iguales los valores numéricos al compararse con un número representado como `str`.

> **!** Importante
>
> Importante asegurarse que al realizar comparaciones las condiciones de cada variable sean lo más similares posibles. Por ejemplo, el código anterior se podría comparar perfectamente si realizamos un *casting* a entero: `int(d)`

```
int(d) == a
```

```
True
```

7.2 Desigualdad

Si el operando de la **izquierda** no es igual al operando de la **derecha**, entonces, el resultado es `True`. De lo contrario, es `False`.

Ejemplifiquemos este operador comparando directamente valores literales en lugar de variables.

```
'Diego' != 'Alberto'
```

```
True
```

```
'Diego' != 'Diego'
```

```
False
```

Ahora evalúemos la igualdad entre dos cadenas aparentemente idénticas pero que difieren por el uso de mayúsculas.

```
'Alberto' != 'alberto'
```

```
True
```

Python no considera igual el caracter `A` y la `a`. Por lo tanto, las cadenas anteriores son diferentes. Nuevamente, es importante tener cuidado al realizar comparaciones que los operandos se encuentren en condiciones similares.

También tomar en consideración que cualquier caracterer adicional, podría afectar nuestra comparación. Por ejemplo, veamos que si comparamos dos cadenas iguales pero colocamos un espacio adicional en una de ellas, entonces, los `str` se considerarían **no iguales**.

```
'Diego' != 'Diego '
```

```
True
```

7.3 Menor que

Si el operador de la **izquierda** es menor que el de la **derecha**, entonces, el resultado es `True`. De lo contrario, es `False`.

7.4 Mayor que

Si el operador de la **izquierda** es mayor que el de la **derecha**, entonces, el resultado es `True`. De lo contrario, es `False`.

Para los siguientes ejemplos, definiremos variables que contienen la edad de tres personas.

```
edad_1 = 7
edad_2 = 16
edad_3 = 33
```

Ahora, comparemos las edades de las personas. Primero, evaluamos si la edad de la persona 1 es menor yor que la edad de la persona 2.

```
edad_1 < edad_2
```

```
True
```

Luego, evaluemos si la edad de la persona 3 es mayor que la de la persona 2.

```
edad_3 > edad_2
```

```
True
```

7.5 Menor o igual que

Si el operador de la **izquierda** es menor o igual que el de la **derecha**, entonces, la condición es `True`. De lo contrario, es `False`.

7.6 Mayor o igual que

Si el operador de la **izquierda** es mayor o igual que el de la **derecha**, entonces, la condición es `True`. De lo contrario, es `False`.

A continuación, crearemos variables con notas de estudiantes.

```
notaLuis = 5.9
notaMaria = 7.8
notaIsa = 6.0
```

Utilizaremos los operadores para evaluar si los estudiantes aprobaron el curso. Se sabe que la nota mínima debe ser de 6.0

```
notaLuis >= 6.0
```

```
False
```

```
notaMaria>= 6.0
```

```
True
```

```
notaIsa>= 6.0
```

```
True
```

> **Advertencia**
>
> Los comparadores anteriores se utilizan para realizar evaluar rangos de valores numéricos. No se recomienda su uso para comparar cadenas de caracteres.

Veamos la siguiente comparación.

```
'100' > '42'
```

```
False
```

Del ejemplo anterior, podemos observar que las comparaciones de números representados por `str` no funcionan de la misma forma que con valores numéricos en `int` o `float`.

Incluso podríamos intentar hacer comparaciones entre textos. Por curiosidad, veamos qué sucede si comparamos cadenas de caracteres.

```
'pequeño' > 'grande'
```

```
True
```

El resultado parece no tener ningún sentido lógico, sin embargo, Python no es capaz de reconocer el significado de un `str`. Realmente, al comparar cadenas de caracteres de esta manera lo que se realiza es una comparación lexicográfica. Es decir, se compara la representación numérica de cada caracter.

Por ejemplo, el caracter `a` se representa por el valor decimal 97, mientras que el caracter `b` se representa por el valor decimal 98. Por eso al realizar la siguiente comparación obtendremos `True`. Ya que es equivalente a comparar `97 < 98`.

```
'a' < 'b'
```

```
True
```

¿Qué sucede cuando tenemos una cadena con más de un caracter? En este caso, se realiza la misma comparación mencionada anteriormente pero, caracter por caracter, hasta encontrar un caracter distinto que se pueda evaluar. Por eso al comparar si `chow chow` es mayor a `chihuahua` obtendremos `True`, ya que `o` (111) es mayor a `i` (105).

```
 'chow chow' > 'chihuahua'
```

```
True
```

> **Advertencia**
>
> En general, no se suelen realizar comparaciones de rangos para `str` y se sugiere utilizarlos exclusivamente para valores numéricos.

> **Nota**
>
> El ejemplo anterior no demuestra que Chow chow sea superior a Chihuahua. El procesador se limita a ejecutar instrucciones ariméticas y lógicas, no tiene conocimientos sobre razas caninas.

8 Operadores Lógicos

Los operadores lógicos se utilizan para evaluar combinaciones lógicas. Generalmente, se utilizan en conjunto con los operadores de comparación para evaluar varias condiciones. Su funcionamiento es idéntico a las *Tablas de Verdad* que se suelen aprender en cursos de Matemáticas.

Los operadores disponibles son:

1. `and`
2. `or`
3. `not`

8.1 and

Si ambos operandos son `True`, entonces, el resultado es `True`. También se podría interpretar de la siguiente manera: si por lo menos uno de los operandos es `False`, entonces, el resultado es `False`. Corresponde al operador de **Conjunción**. Su tabla de verdad es la siguiente.

a	b	a and b
False	False	False
False	True	False
True	False	False
True	True	True

Iniciemos creando variables para representar la calificación de tres películas.

```
the_dark_knight = 9.0
interstellar = 8.7
inception = 8.8
```

Si quisieramos evaluar si `the_dark_knight` tiene una puntuación mayor que las otras dos, se podría sugerir realizar la comparación de la siguiente manera lógica.

```
the_dark_knight > interstellar > inception
```

```
False
```

Sin embargo, no obtendríamos el resultado esperado. En este caso, al concatenar el operador de comparación `>`, Python evalúa que `the_dark_knight` sea mayor que `interstellar`, y luego, que `interstellar` sea mayor que `inception`. Por esta razón obtenemos `False`.

Por lo tanto, si lo que queremos evaluar realmente es si el valor de `the_dark_knight` es mayor a los otros dos, entonces, podríamos utilizar el operador `and` de la siguiente manera.

```
the_dark_knight > interstellar and the_dark_knight > inception
```

```
True
```

A continuación, realizaremos las operaciones de la línea anterior de forma independiente para comprender cómo funciona realmente. Primero, se realizan las operaciones comparativas. Observamos que el resultado de cada comparación devuelve un resultado `True`.

```
the_dark_knight > interstellar
```

```
True
```

```
the_dark_knight > inception
```

```
True
```

Luego, se realiza la operación lógica. En este caso, realizaremos la operación `and` directamente con los valores que nos devolvieron las comparaciones anteriores.

```
True and True
```

```
True
```

El resultado coincide con lo mostrado en la tabla de verdad para la conjunción.

> **Precaución**
>
> Nótese que es necesario realizar la comparación contra cada película de forma independiente. Un error bastante común consiste en realizar la operación de la siguiente manera, lo cual produciría un resultado incorrecto o carente de sentido lógico.

```
the_dark_knight > interstellar and inception
```

```
8.8
```

En la expresión anterior, se inicia realizando la comparación.

```
the_dark_knight > interstellar
```

```
True
```

Pero luego, al evaluar el operador lógico, realmente se está realizando la siguiente expresión.

```
True and inception
```

```
8.8
```

> **Nota**
>
> Utilizando el operador **and** se pueden añadir tantas comparaciones como sea necesario, por ejemplo, si necesitaramos comparar contra más de dos películas. Lo importante es tener presente que el resultado será `True` sí y solo sí **todos** los operandos son `True`.

```
the_prestige = 8.5

(the_dark_knight > interstellar
and the_dark_knight > inception
and the_dark_knight > the_prestige)
```

```
True
```

> **Nota**
>
> Si se requiere separar un código de Python en múltiples líneas, se puede encerrar la expresión completa dentro de paréntesis `()`.

8.2 or

Si al menos uno de los operandos es `True`, entonces, el resultado es `True`. En este caso, el resultado será `False`únicamente cuando todos los operandos sean `False`. Su tabla de verdad corresponde al operador de Disyunción.

a	b	a or b
False	False	False
False	True	True
True	False	True
True	True	True

Crearemos la siguiente variables para comprender su funcionamiento. Asumamos que una tienda tiene una promoción especial del 20% de descuento si el cliente realiza el pago con una tarjeta de crédito a 6 o 12 cuotas.

```
numCuotas = 6
```

Entonces, podríamos utilizar el operador `or` para evaluar si la cantidad de cuotas seleccionada por el cliente, `numCuotas`, coincide con el valor 6 o 12.

```
numCuotas == 6 or numCuotas == 12
```

```
True
```

Al evaluar la expresión, obtenemos el resultado de `True` debido a que la primera comparación es verdadera. Al igual que la subsección anterior, analicemos cómo es evaluada la expresión anterior en pasos.

Primero, se realizan las operaciones de comparación.

```
numCuotas == 6
```

```
True
```

```
numCuotas == 12
```

```
False
```

Luego, se toman los resultados de cada comparación y se evalúa el operador lógico.

```
True or False
```

```
True
```

El resultado coincide con la tabla de verdad para disyunción.

> **Nota**
>
> Al igual que el operador `and`, es posible anidar más comparaciones en una misma expresión. Por ejemplo, podríamos ampliar la promoción a 18 y 24 cuotas.

```
(numCuotas == 6
 or numCuotas == 12
 or numCuotas == 18
 or numCuotas == 24)
```

```
True
```

8.3 not

El operador `not` invierte el valor lógico del operando. Si el operando es `True`, entonces, el resultado es `False`. Y si el operando es `False`, entonces, el resultado será `True`. Corresponde a la tabla de la negación.

a	not a
False	True
True	False

Inicialmente, podría parecer que este operador no tiene ninguna utilidad ya que simplemente invierte el valor de un operando. Sin embargo, en ciertas ocasiones podría resultar conveniente utilizarlo, especialmente, para hacer más clara (y natural) una expresión.

A continuación, supongamos que tenemos las siguientes variables que indican si una persona si se encuentra en línea y si está ocupada.

```
enLinea = True
ocupada = False
```

Para determinar si la persona se encuentra disponible para contactarla, podríamos escribir la siguiente expresión. Si su resultado es `True`, entonces, contactamos a la persona, si es `False`, entonces, no.

```
enLinea and not ocupada
```

```
True
```

La línea de código se lee con bastante naturalidad: la contactaré si "está en línea y no está ocupada". Claramente, se ve más legible a que hubiéramos utilizado una línea como la siguiente.

```
enLinea==True and ocupada==False
```

```
True
```

> **Tip**
>
> Aunque la expresión `enLinea==True` es correcta, nótese que daría el mismo resultado que simplemente escribir el nombre de la variable ya que si su valor fuera `True`, el resultado sería equivalente a `True===True`, que daría `True`. Si el valor de la variable fuera `False`, entonces, se evalúa `False==True`, dando como resultado `False`. En conclusión, se obtendría siempre el mismo valor de la variable original.

8.4 Precedencia

Es bastante común utilizar una combinación de operadores lógicos en una misma expresión. Tomar en consideración que los operadores presentan el siguiente orden de precedencia:

0. `()`
1. `not`
2. `and`
3. `or`

Por lo tanto, se debe tener especial cuidado al emplear distintos operadores lógicos en una misma línea para obtener el resultado deseado. Se recomienda el uso de paréntesis para indicar las operaciones iniciales, y también, para mejorar la claridad del código.

Finalmente, realicemos un ejemplo en el que evalúemos una expresión más compleja. Iniciaremos definiendo las siguientes variables sobre condiciones climáticas.

```
llueve = False
estado = 'nublado'
temperatura = 20
```

Crearemos una expresión que nos ayudará a decidir si es un momento conveniente para salir a caminar. Iniciemos definiendo cuáles serían las condiciones favorables utilizando lenguaje común: saldré a caminar si no llueve y si el estado del clima es nublado o soleado con una temperatura menor a los 24 grados Celcius.

```
not llueve and estado=='nublado' or estado=='soleado' and temperatura<24
```

```
True
```

La expresión anterior podría funcionar correctamente para algunos casos, sin embargo, si evalúamos el siguiente escenario veremos que el resultado no es el esperado.

```
llueve = True
estado = 'soleado'
temperatura = 18

not llueve and estado=='nublado' or estado=='soleado' and temperatura<24
```

```
True
```

El código nos indica que podemos salir, sin embargo, nos interesa que el programa indique `False` cuando haya lluvia. Antes de mostrar una posible solución al problema, veamos cómo evalúa Python la expresión planteada y descubramos por qué no funciona en este caso.

Iniciamos reemplazando cada variable con su valor correspondiente.

```
not True and 'soleado'=='nublado' or 'soleado'=='soleado' and 18<24
```

Evalúemos los operadores de comparación para obtener los resultados `bool`.

```
not True and False or True and True
```

Respetando el orden de precedencia de los operadores lógicos. Iniciamos evaluando el `not`.

```
False and False or True and True
```

Continúamos evaluando el operador `and`.

```
False or True
```

Finalizamos evaluando el operador “‘or“‘.

```
True
```

Ahora vemos que el orden de precedencia ha afectado el funcionamiento de nuestro código. En este caso, se sugiere colocar paréntesis `()` para indicar el orden en el que nos interesa realizar las operaciones lógicas.

```
not llueve and (estado=='nublado' or (estado=='soleado' and temperatura<24))
```

```
False
```

De esta manera, se ha solucionado el error presentado anteriormente. Además, hemos mejorado la legibilidad de nuestro código significativamente, ahora se puede distingir claramente cuáles son las condiciones que se deben cumplir para salir a caminar.

9 Sentencias condicionales

Las sentencias condicionales modifican la ejecución del programa. Se utilizan para automatizar la toma de decisiones.

En Python, podemos utilizar las siguientes sentencias condicionales:

- if
- elif
- else

9.1 if

Si la condición es `True`, entonces, se ejecuta el bloque de código con indentación siguiente. Si la condición es `False`, entonces, salta el bloque de código con indentación y continúa la ejecución del programa.

La palabra `if` se utiliza para iniciar una o más sentencias condicionales (que veremos más adelante).

```
if expresion:
  codigo
```

En lenguaje humano, esta sentencia se leería de la siguiente manera: "Si la expresión es verdadera, entonces, ejecutar código"

9.1.1 Ejemplo

Por ejemplo, definimos `monto` para almacenar el valor de un producto y `aplicarDescuento` para especificar si se aplicará el descuento o no. Luego, colocamos la palabra `if` seguida por la variable `aplicarDescuento`. Si la expresión es `True`, entonces, se ejecutará el bloque de código conformado por las tres líneas indentadas (con espacios). Si la expresión resulta ser `False`, entonces, el programa se saltará las líneas con indentación y continuará con el resto del código.

Asignemos un valor `False`a la variable `aplicarDescuento` y veamos qué se sucede.

```
monto = 50.0
aplicarDescuento = False

if aplicarDescuento:
  descuento = 0.2*monto
  monto = monto - descuento
  print(f'Descuento: Q{descuento}')
print(f'Monto a cancelar: Q{monto}')
```

```
Monto a cancelar: Q50.0
```

En este caso, al evaluar la sentencia condicional se obtiene `False`, por lo tanto, el código con indentación siguiente **no es ejecutado** y se ejecuta el código siguiente que ya no se encuentra indentado. Por ello, solamente se imprime el monto a cancelar. Nótese que la variable `monto` mantiene el valor original asignado en la primera línea del programa.

Ahora modifiquemos el valor de `aplicarDescuento` a `True`.

```
monto = 50.0
aplicarDescuento = True

if aplicarDescuento:
  descuento = 0.2*monto
  monto = monto - descuento
  print(f'Descuento: Q{descuento}')
print(f'Monto a cancelar: Q{monto}')
```

```
Descuento: Q10.0
Monto a cancelar: Q40.0
```

Ahora, al evaluar la sentencia condicional se obtiene `True`, por lo tanto, se ejecuta el bloque de código siguiente con indentación. Es decir, se ejecutan las tres líneas de código en las cuales se crea la variable `descuento`, se modifica el `monto` y se imprime el descuento aplicado. Al concluir el código dentro de la condicional, se continúa con la ejecución normal del programa. Por ello, al igual que el caso anterior, siempre se imprime el monto a cancelar al final.

9.2 elif

Si la sentencia anterior (`if` o `elif`) no se cumple, entonces, se evalúa esta sentencia.

En una serie de sentencias, se puede agregar uno o múltiples `elif` para evaluar diversas condiciones.

> **Nota**
>
> Una serie de sentencias siempre inicia por `if`. Por lo tanto, no se podría utilizar `elif` de forma independiente.

```
if expresion1:
  codigo1
elif expresion2:
  codigo2
elif expresion3:
  codigo3
```

En lenguaje humano, esta sentencia se leería: "Si la expresión 1 es verdadera, entonces, ejecutar código 1. Sino, si la expresión 2 es verdaera, entonces, ejecutar código 2. Sino, si la expresión 3 es verdadera, entonces, ejecutar código 3". Y podríamos continuar así, sucesivamente, con todas las expresiones necesarias.

9.2.1 Ejemplo

El siguiente ejemplo, crearemos un programa para determinar si un estudiante ha aprobado un curso. La nota mínima para aprobar es de 60 puntos, por lo tanto, si la nota del estudiante es mayor o igual o 60, entonces, aprueba, sino, reprueba.

Iniciemos asignando un valor de 90 a la nota del estudiante y ejecutemos el programa.

```
notaEstudiante = 90
notaMinima = 60

if notaEstudiante >= notaMinima:
  print(f'Has aprobado el curso con una nota de {notaEstudiante}')
  print('¡Felicidades!')
elif notaEstudiante < notaMinima:
  print(f'Has reprobado el curso con una nota de {notaEstudiante}')
print('Gracias por utilizar el programa')
```

```
Has aprobado el curso con una nota de 90
¡Felicidades!
Gracias por utilizar el programa
```

Como es de esperarse, se muestra que el estudiate ha aprobado el curso. Analicemos paso a paso la ejecución del programa para comprender el "flujo de ejecución". Primero, se asignan valores a las variables `notaEstudiante` y `notaMinima`. Luego, se evalúa si nota del estudiante es mayor o igual a la nota mínima. Debido a que esta expresión es verdadera, entonces, se ejecuta el código con indentación siguiente. Finalmente, como la condición del `if` se cumplió, entonces, el programa **no** evalúa las siguientes sentencias sino que se mueve hasta el final de las sentencias para continuar con la ejecución del resto del código.

Ahora, modifiquemos la nota del estudiante para que aparezca como reprobado.

```
notaEstudiante = 55
notaMinima = 60

if notaEstudiante >= notaMinima:
  print(f'Has aprobado el curso con una nota de {notaEstudiante}')
  print('¡Felicidades!')
elif notaEstudiante < notaMinima:
  print(f'Has reprobado el curso con una nota de {notaEstudiante}')
print('Gracias por utilizar el programa')
```

```
Has reprobado el curso con una nota de 55
Gracias por utilizar el programa
```

El código inicia evalúando si la nota del estudiante es mayor o igual a la nota mínima. Ahora la expresión es falsa, por lo tanto, no ejecuta el bloque de código siguiente sino que se dirige a la siguiente sentencia condicional. Ahora en el `elif` evalúa si la nota del estudiante es menor a la nota mínima. Debido a que la expresión es verdadera, entonces, se ejecuta la línea de código siguiente indicando que se ha reprobado el curso. Finalmente, se imprime el mensaje de despedida, continuando con la ejecución del resto del programa.

9.3 else

Si ninguna de las condiciones anteriores se cumple, entonces, se ejecuta el código dentro del **else**.

Es la única sentencia que **no requiere de una condición para evaluar**.

```
edadJohn = 18
edadIsa = 18

if edadJohn > edadIsa:
```

```
  print("John es mayor a Isa")
elif edadJohn < edadIsa:
  print("Isa es mayor a John")
else:
  print("Ambos tienen la misma edad")
```

```
Ambos tienen la misma edad
```

9.4 Flujo de control

A continuación se muestran algunos ejemplos de la utilización de las sentencias condicionales para comprender cómo afecta la ejecución del programa.

9.4.1 Ejemplo 1

En este ejemplo evalúamos si un número es par o impar. Veremos que podemos emplear **if** y **else** sin necesidad de colocar un **elif**.

```
num = 7

if num%2 == 0:
  print(num, "es par")
else:
  print(num, "es impar")
```

```
7 es impar
```

9.4.2 Ejemplo 2

El siguiente código, evalúa el índice de masa corporal de una persona e imprime un mensaje dependiendo del resultado.

```
pesoKg = 100
alturaM = 1.7

imc = pesoKg/alturaM**2
imc = round(imc, 2)
print("Su resultado de IMC es", imc)
```

```
if imc < 15:
  print("Bajo peso")
elif imc < 25:
  print("Peso normal")
elif imc < 30:
  print("Sobrepeso")
else:
  print("Obesidad")

print("Fin")
```

```
Su resultado de IMC es 34.6
Obesidad
Fin
```

El siguiente código no utiliza adecuadamente las sentencias condicionales.

```
pesoKg = 40
alturaM = 1.7

imc = pesoKg/alturaM**2
imc = round(imc, 2)
print("Su resultado de IMC es", imc)

if imc < 15:
  print("Bajo peso")
if imc < 25:
  print("Peso normal")
if imc < 30:
  print("Sobrepeso")
else:
  print("Obesidad")

print("Fin")
```

```
Su resultado de IMC es 13.84
Bajo peso
Peso normal
Sobrepeso
Fin
```

9.4.3 Ejemplo 3

En el siguiente ejemplo, se emplean operadores de comparación en conjunto con operadores lógicos. Esto nos permite unir más de una expresión dentro de la misma condición.

```
print("¿Cuál es tu año de nacimiento? ")
birthDate = 1995

if birthDate>=1944 and birthDate<=1964:
  generation = "Baby Boomer"
elif birthDate>=1965 and birthDate<=1979:
  generation = "X"
elif birthDate>=1980 and birthDate<=1994:
  generation = "Y o millennial"
elif birthDate>=1995 and birthDate<=2015:
  generation = "Z"
else:
  generation = "Desconocida"

print("Perteneces a la generación", generation)
```

```
¿Cuál es tu año de nacimiento?
Perteneces a la generación Z
```

9.5 Método abreviado para if-else

Existe una forma de abreviar códigos sencillos en los cuales se utiliza if-else y solamente poseen una línea de indentación.

Por ejemplo, si tenemos el siguiente programa que evalúa si un cliente debe obtener una pizza gratis o no.

La condición es que si la pizza fue entregada un tiempo mayor a 30 minutos, entonces la pizza es gratis. De lo contrario, se indica el monto a cancelar.

```
entrega = 18

if entrega <= 30:
  print("Total: Q99")
else:
  print("Gratis")
```

```
Total: Q99
```

El código de las sentencias condicionales se puede resumir en una sola línea, cambiando el orden de la siguiente forma:

```
entrega = 45

print("Total: Q99" if entrega <= 30 else "Gratis")
```

```
Gratis
```

10 Búcles

Los búcles se utilizan para repetir la ejecución un conjunto de instrucciones (iteración).

Existen dos tipos de búcles: * while * for

10.1 while

El búcle *while* permite ejecutar un conjunto de instrucciones siempre que la condición sea **True**.

Utiliza la siguiente *sintaxis*:

```
while condición:
  instrucción1
  instrucción2
  ...
  instrucciónN
```

```
pizza = 4
while pizza > 0:
  print("Porción entregada.")
  pizza -= 1

print("No quedan más porciones.")
```

```
Porción entregada.
Porción entregada.
Porción entregada.
Porción entregada.
No quedan más porciones.
```

10.1.1 break

Detiene la ejecución de un búcle, aunque la condición sea verdadera.

```
acceso = False #flag
while acceso == False:
  ingreso = "5426" #input("Código de acceso: ")
  if ingreso == "5426":
    print("Acceso concedido.")
    acceso = True
  else:
    print("Código inválido.")

print("Ingresando al sistema.")
```

```
Acceso concedido.
Ingresando al sistema.
```

```
while True:
  ingreso = "5426" #input("Código de acceso: ")
  if ingreso == "5426":
    print("Acceso concedido.")
    break
  else:
    print("Código inválido.")

print("Ingresando al sistema.")
```

```
Acceso concedido.
Ingresando al sistema.
```

10.1.2 continue

Salta la iteración actual y continúa con la siguiente.

```
number = '124,256,312'
result = ''
index = 0

while index<len(number):
```

```
  if number[index]==',':
    index += 1
    continue
  result += number[index]
  index += 1

print(result)
```

```
124256312
```

```
result = number.replace(',','')
print(result)
```

```
124256312
```

10.1.3 Ejemplo

A continuación, crearemos un juego sencillo en el cual seleccionaremos un número al azar y el usuario deberá adivinar cuál es el número.

```
import random

number = random.randint(1,9)
guess = 0

print("Welcome to Guess a Number")
while guess != number:
  print("\nChoose a number from 1 to 9: ")
  print("Write -1 if you want to exit.")
  #guess = int(input())
  print(f"{guess=}")
  if guess==-1:
    break
  if guess>9:
    print("Your number is too big.")
    continue
  if guess==number:
    print("Congratulations!")
    break
  else:
```

```
    print("Try again.")
    guess += 1
print("Thank you for playing")
```

```
Welcome to Guess a Number

Choose a number from 1 to 9:
Write -1 if you want to exit.
guess=0
Try again.

Choose a number from 1 to 9:
Write -1 if you want to exit.
guess=1
Try again.

Choose a number from 1 to 9:
Write -1 if you want to exit.
guess=2
Try again.

Choose a number from 1 to 9:
Write -1 if you want to exit.
guess=3
Try again.

Choose a number from 1 to 9:
Write -1 if you want to exit.
guess=4
Try again.

Choose a number from 1 to 9:
Write -1 if you want to exit.
guess=5
Try again.
Thank you for playing
```

10.2 for

El búcle for se utiliza para realizar iteraciones sobre una secuencia (lista, tupla, diccionario, conjunto, string).

Se ejecuta el conjunto de instrucciones hasta iterar sobre la secuencia completa.

```
pares = [0, 2, 4, 6, 8]
for n in pares:
  print(n)
  print('es par.')

print('Búcle finalizado')
```

```
0
es par.
2
es par.
4
es par.
6
es par.
8
es par.
Búcle finalizado
```

```
nombre = 'Diego'
for i in nombre:
  print(i)
```

```
D
i
e
g
o
```

10.2.1 range()

La función *range()* retorna una secuencia de números iniciando en 0 (por defecto) e incrementando en 1 (por defecto) hasta el número indicado (sin incluirlo).

Utiliza la siguiente sintaxis.

```
range(inicio, final, paso)
```

```
for x in range(0, 10):
  print(x)
```

```
0
1
2
3
4
5
6
7
8
9
```

10.2.2 break

Detiene la ejecución del búcle aunque no se haya iterado por toda la secuencia.

```
personas = [0, 0, 0, 1, 0, 0]

for persona in personas:
  if persona==1:
    break
  print('Pato...')
print('¡Ganso!')
```

```
Pato...
Pato...
Pato...
¡Ganso!
```

10.2.3 continue

Salta la iteración actual y continúa con la siguiente.

```
limite = 47
for x in range(limite):
    if x % 2 == 0:
        continue
    print(x)
```

```
1
3
5
7
9
11
13
15
17
19
21
23
25
27
29
31
33
35
37
39
41
43
45
```

10.2.4 Ejemplo

A continuación, utilizaremos un búcle for para crear un programa de cifrado.

Recordemos que los string son cadenas de caracteres. Cada caracter se puede representar por un número (ver ASCII Chart).

Podemos utilizar la función *ord()* para determinar el valor decimal de un caracter. También podemos utilizar la función *chr()* para determinar el caracter correspondiente a un valor decimal.

```
print(ord('A'))
```

```
65
```

```
print(chr(68))
```

```
D
```

El siguiente código utiliza un búcle *for* para codificar un texto a partir de una llave k.

```
texto = "Me gusta programar"
k = 3
cifrado = ""
for letra in texto:
    x = ord(letra)
    x = x + k
    c2 = chr(x)
    cifrado += c2
print(cifrado)
```

```
Ph#jxvwd#surjudpdu
```

El siguiente realiza el procedimiento inverso. En este caso, se ingresa un texto cifrado y se decodifica a partir de la llave correcta.

```
cifrado = 'Ph#jxvwd#surjudpdu'
k = 3
decodificacion= ""
for c in cifrado:
    x = ord(c)
    x = x - k
    c2 = chr(x)
    decodificacion += c2
print(decodificacion)
```

```
Me gusta programar
```

11 String

Un string es una cadena o secuencia de caracteres (letras, números o símbolos).

11.1 Inicialización

Para crear un string, simplemente colocamos la cadena de caracteres dentro de comillas simples o dobles.

```
saludo = '¡Bienvenido a mi programa! _versión 2.1_'
print(saludo)
```

```
¡Bienvenido a mi programa! _versión 2.1_
```

También podemos utilizar la función **str()** para convertir cualquier tipo de dato a string.

```
usuario = 27
print(type(usuario))
usuario = str(usuario)
print(type(usuario))
```

```
<class 'int'>
<class 'str'>
```

11.2 Concatenación

Podemos unir dos strings utilizando el símbolo +. Asegurarse que ambos valores sean de tipo *str*.

```
message = "Your name is "
name = "Diego"
text = message+name
print(text)
```

```
Your name is Diego
```

```
a = "El precio es de: "
b = 19.99
c = a+str(b)
print(c)
```

```
El precio es de: 19.99
```

11.3 Longitud

Podemos determinar el tamaño del string, es decir, el número de caracteres utilizando la función **len()**.

```
ciudad = "Guatemala"
longitud = len(ciudad)
print(longitud)
```

```
9
```

11.4 Indexación

Podemos acceder a caracteres individuales utilizando el operador de indexación [ ] justo después del string.

Dentro de paréntesis, se debe especificar el índice del caracter.

Importante: Los índices inician con *0*.

Para acceder al último caracter, se puede colocar el índice *-1*

Por ejemplo, si tenemos un string "python", estos serían los índices de cada caracter.

0	1	2	3	4	5
p	y	t	h	o	n

```
x = "python 3"
print(x)
```

python 3

```
print(x[0])
```

```
p
```

```
print(x[-1])
```

```
3
```

11.5 Corte (slicing)

Podemos acceder a grupos de caracteres utilizando el operador de indexación y especificando el índice inicial y final separados por dos puntos [:].

Obtendremos un string conformado por todos los caracteres del índice de inicio hasta el índica del final (sin incluir el este último).

Si se omite el índice al inicio, se selecciona por defecto el índice 0. Y si se omite el índice del final, se selecciona por defecto el último índice.

Por ejemplo, si tenemos el siguiente string.

0	1	2	3	4	5	6	7	8	9	10	11	12
D	i	e	g	o		M	o	r	a	l	e	s

```
nombreCompleto = "Diego Morales"
print(nombreCompleto)
```

```
Diego Morales
```

```
nombre = nombreCompleto[:5]
print(nombre)
```

```
Diego
```

```
apellido = nombreCompleto[6:]
print(apellido)
```

```
Morales
```

11.6 Métodos

Los string poseen métodos integrados que podemos utilizar. Estos métodos no modifican el string original sino que retornan un nuevo string con la salida.

Documentación de Python: String Methods

```
date = "   FECHA: 1 de enero de 1970 "
print(date)
```

```
   FECHA: 1 de enero de 1970
```

```
date = date.strip()
print(date)
```

```
FECHA: 1 de enero de 1970
```

```
"enero" == "Enero".lower()
```

```
True
```

```
print(date.upper())
```

```
FECHA: 1 DE ENERO DE 1970
```

```
print(date.lower())
```

```
fecha: 1 de enero de 1970
```

```
year = "2020"
print(year)
print(year.isnumeric())
```

```
2020
True
```

```
print("enero" in date)
```

```
True
```

```
print(date.find("enero"))
```

```
12
```

```
print(date.find("marzo"))
```

```
-1
```

```
print(date.replace("enero","febrero"))
```

```
FECHA: 1 de febrero de 1970
```

```
elementos = date.split()
print(elementos)
print(elementos[3])
```

```
['FECHA:', '1', 'de', 'enero', 'de', '1970']
enero
```

11.6.1 Ejemplo

Recibe un string con información sobre una dirección de correo electrónico. Se le solicita extraer el correo y luego extraer el usuario del mismo.

```
mensaje = "Correo de: diego.morales@ameritec.edu.gt  "
print(mensaje)
```

```
Correo de: diego.morales@ameritec.edu.gt
```

```
start = mensaje.find(":")
print(start)
```

```
9
```

```
correo = mensaje[start+1:]
print(correo)
```

```
 diego.morales@ameritec.edu.gt
```

```
correo = correo.strip()
print(correo)
```

```
diego.morales@ameritec.edu.gt
```

```
partes = correo.split("@")
print(partes)
```

```
['diego.morales', 'ameritec.edu.gt']
```

```
usuario = partes[0]
print(usuario)
```

```
diego.morales
```

11.7 Formato de cadenas

En Python 3 podemos utilizar un formato para los string que mezclan caracteres con variables.

Se debe inicializar el string escribiendo una *f* y especificar dentro de llaves las variables *{ }*.

```python
estudiante = "Diego"
actividad = 3
nota = 100.0

mensaje = f"Hola {estudiante}, tu nota de la actividad {actividad} es de {nota}"
print(mensaje)
```

```
Hola Diego, tu nota de la actividad 3 es de 100.0
```

12 Estructuras de Datos

Permiten organizar, manipular y almacenar objetos relacionados para que se pueda trabajar con ellos de forma eficiente. Se emplea una única variable para almacenar varios objetos.

Existen las siguientes estructuras de datos en Python:

- Listas (List)
- Tuplas (Tuples)
- Conjuntos (Set)
- Diccionarios (Dictionaries)

12.1 Intuición sobre Listas

Imagine que se le entrega la siguiente lista de compras.

Índice	Frutas
0	Manzana
1	Banano
2	Naranja
3	Mango

Cuando usted llega al supermercado, se da cuenta de que las naranjas están agotadas, pero sí hay mandarinas. Entonces, usted se dirige al indice 2, y escribe *Mandarina* en su lugar.

Índice	Frutas
0	Manzana
1	Banano
2	Mandarina
3	Mango

Luego de recoger todos los artículos, se da cuenta de que esta semana el mango está muy caro y prefiere no comprarlo, así que lo borra de la lista.

Índice	Frutas
0	Manzana
1	Banano
2	Mandarina

Finalmente, cuando llega a la caja, le ofrecen pera y kiwi en oferta y usted decide agregarlos al listado y llevárselos. Su lista resultante, sería la siguiente:

Índice	Frutas
0	Manzana
1	Banano
2	Mandarina
3	Pera
4	Kiwi

Las listas en Python funcionan de forma similar, permitiendo agrupar artículos similares (frutas, en este caso) y permitiendo realizar modificaciones sobre ellas (cambiar, eliminar, agregar).

12.2 Listas

- Permiten almacenar cualquier tipo de objeto (no necesariamente deben ser todos de la misma clase).
- Son modificables (mutables).
- Dinámicas

12.2.1 Creación de Listas

Se emplea la siguiente sintaxis.

```
nombreLista = [elemento0, elemento1, elemento2, ...]
```

```
frutas = ['manzana', 'pera', 'naranja', 'mora']
print(frutas)
```

```
['manzana', 'pera', 'naranja', 'mora']
```

```
notas = [90, 89.5, 92, 95.5]
print(notas)
```

```
[90, 89.5, 92, 95.5]
```

12.2.2 Indexación

Para acceder a elementos de listas se utiliza el operador de indexación y se especifica el índice del objeto [index].

```
print(frutas[3])
```

```
mora
```

```
print(frutas[-1])
```

```
mora
```

```
print(frutas[1:3])
```

```
['pera', 'naranja']
```

12.2.3 Asignación

Para asignar un nuevo valor a una posición de una lista, se especifica el elemento (o elementos) que se desean modificar accediendo a él y se le asigna su nuevo valor.

```
print(notas)
```

```
[90, 89.5, 92, 95.5]
```

```
notas[2] = 100
```

```
notas[0:2] = [85, 87]
```

12.2.4 Agregar y remover elementos

Se pueden agregar elementos al final de una lista utilizando **append()** y se puede remover algún elemento en específico utilizando **remove()**. Dentro del paréntesis se coloca el objeto que se desea agregar o eliminar.

Observación: Si se intenta remover un objeto que no está en la lista, se producirá un error.

```
print(frutas)
```

```
['manzana', 'pera', 'naranja', 'mora']
```

```
frutas.append("fresa")
```

```
frutas.remove("pera")
```

12.2.5 Número de elementos

Para obtener el número de elementos en una lista se utiliza el método **len()**.

```
numeroFrutas = len(frutas)
```

```
print("La lista contiene",numeroFrutas,"frutas.")
```

```
La lista contiene 4 frutas.
```

12.2.6 Existencia del elemento

Para verificar si un elemento se encuentra dentro de una lista se utiliza in.

El resultado será **True** si el elemento se encuentra en la lista y **False** de no ser así.

```
print("cereza" in frutas)
```

```
False
```

```
print("fresa" in frutas)
```

```
True
```

12.2.7 Recorrer elementos

Podemos emplear un búcle **for** para recorrer por los elementos de una lista.

```
for x in notas:
  print(x)
```

```
85
87
100
95.5
```

```
for i in range(len(notas)):
  print(i, frutas[i])
```

```
0 manzana
1 naranja
2 mora
3 fresa
```

12.3 Tuplas

- Permiten almacenar cualquier tipo de objeto (no necesariamente deben ser todos de la misma clase).
- No son modificables (inmutables).
- Más rápidas que las listas.
- Protección contra escritura.
- Se suelen utilizar con diccionarios.

Se emplea la siguiente sintaxis.

```
nombreTupla = (elemento0, elemento1, elemento2, ...)
```

```
frutas = ('manzana', 'pera', 'naranja', 'mora')
print(frutas)
```

```
('manzana', 'pera', 'naranja', 'mora')
```

```
notas = (90, 89.5, 92, 95.5)
print(notas)
```

```
(90, 89.5, 92, 95.5)
```

12.3.1 Indexación

Para acceder a elementos de una tupla se utiliza el operador de indexación y se especifica el índice del objeto [index].

```
print(frutas[3])
```

```
mora
```

```
print(frutas[-1])
```

```
mora
```

```
print(frutas[1:4])
```

```
('pera', 'naranja', 'mora')
```

12.3.2 Número de elementos

Para obtener el número de elementos en una tupla se utiliza el método **len()**.

```
numeroFrutas = len(frutas)
```

```
print("La tupla contiene",numeroFrutas,"frutas.")
```

```
La tupla contiene 4 frutas.
```

12.3.3 Existencia del elemento

Para verificar si un elemento se encuentra dentro de una tupla se utiliza **in**.

El resultado será **True** si el elemento se encuentra en la tupla y **False** de no ser así.

```
print("cereza" in frutas)
```

```
False
```

```
print("naranja" in frutas)
```

```
True
```

12.3.4 Recorrer elementos

Podemos emplear un búcle **for** para recorrer por los elementos de una tupla.

```
for x in notas:
  print(x)
```

```
90
89.5
92
95.5
```

```
for i in range(len(notas)):
  print(i, notas[i])
```

```
0 90
1 89.5
2 92
3 95.5
```

12.4 Conjuntos

- Permiten almacenar cualquier tipo de objeto.
- Implementa operaciones de teoría de conjuntos.
- No se puede acceder a los elementos.
- No se pueden modificar los elementos existentes, pero sí se permite añadir y eliminar.
- Los elementos no tienen un orden específico.
- No se pueden repetir los elementos dentro del cojunto (son únicos).

Se emplea la siguiente sintaxis.

```
nombreConjunto = {elementoA, elementoB, elementoC, ...}
```

```
companies = {"Apple", "Google", "Facebook", "Amazon"}
print(companies )
```

```
{'Google', 'Amazon', 'Facebook', 'Apple'}
```

12.4.1 Recorrer elementos

Podemos emplear un búcle **for** para recorrer por los elementos de un conjunto.

```
for c in companies:
  print(c)
```

```
Google
Amazon
Facebook
Apple
```

12.4.2 Existencia del elemento

Para verificar si un elemento se encuentra dentro de un conjnuto se utiliza **in**.

El resultado será **True** si el elemento se encuentra en la lista y **False** de no ser así.

```
print("Facebook" in companies)
```

```
True
```

```
print("NVIDIA" in companies)
```

```
False
```

12.4.3 Agregar y remover elementos

Se pueden agregar elementos al final de un conjunto utilizando **add()** y se puede remover algún elemento en específico utilizando **remove()**. Dentro del paréntesis se coloca el objeto que se desea agregar o eliminar.

```
companies.remove('Google')
```

```
companies
```

```
{'Amazon', 'Apple', 'Facebook'}
```

12.4.4 Operaciones de conjuntos

Podemos emplear los siguientes métodos para realizar operaciones entre dos conjuntos.

- union
- intersection
- issubset
- issuperset
- difference

```
a = {1, 2, 3, 4, 5}
b = {2, 4, 6, 8}
c = {4, 5}
```

```
a.union(b)
```

```
{1, 2, 3, 4, 5, 6, 8}
```

```
a.intersection(b)
```

```
{2, 4}
```

```
c.issubset(a)
```

```
True
```

```
b.difference(c)
```

```
{2, 6, 8}
```

12.4.5 Intuición sobre diccionarios

Por ejemplo, imagine que se le solicita crear una estructura de datos para almacenar numeros telefónicos de personas. Si utilizara una lista, el resultado sería el siguiente:

Índice	Número telefónico
0	2222-4466
1	2358-0000
2	2468-0246

En este caso, resultaría muy difícil buscar el número de alguna persona en específico. En cambio, si utilizamos diccionarios, obtendríamos lo siguiente:

Nombre	Número telefónico
William	2222-4466
Emma	2358-0000
John	2468-0246

Ahora, si quisiera obtener el número teléfonico de alguna persona, solo tendría que buscarlo por su nombre.

12.5 Diccionarios

- Permiten almacenar cualquier tipo de objeto asociando un identificador (key) y un valor (value).
- Se pueden modificar los valores de los identificadores.
- Los elementos no tienen un orden específico.

12.5.1 Creación de Diccionarios

Se emplea la siguiente sintaxis.

```
nombreDiccionario = {key0: value0, key1: value1, key2: value2, ...}
```

```
tel = {"William":22224466, "Emma":23580000, "John":24680246}
print(tel)
```

```
{'William': 22224466, 'Emma': 23580000, 'John': 24680246}
```

```
print(tel.keys())
```

```
dict_keys(['William', 'Emma', 'John'])
```

```
print(tel.values())
```

```
dict_values([22224466, 23580000, 24680246])
```

12.5.2 Indexación

Para acceder a elementos de un diccionario se utiliza el operador de indexación y se especifica el identificar o *key* [index].

```
print(tel["William"])
```

```
22224466
```

12.5.3 Asignación de un elemento

Para asignar un valor a un identificar, se especifica el elemento (o elementos) que se desean modificar accediendo a él y se le asigna su nuevo valor.

```
tel["John"] = 22222222
print(tel)
```

```
{'William': 22224466, 'Emma': 23580000, 'John': 22222222}
```

12.5.4 Agregar y remover elementos

Se pueden agregar elementos a un diccionario utilizando el operador de indexación y especificando el nombre del nuevo identificador, luego, se coloca el operador de asignación y se asigna un valor.

Para eliminar un identificar de un diccionario se puede utilizar el método **pop()**.

```
print(tel)
```

```
{'William': 22224466, 'Emma': 23580000, 'John': 22222222}
```

```
tel['Diego'] = 22009999
```

```
tel.pop('Emma')
```

```
23580000
```

12.5.5 Recorrer elementos

Podemos emplear un búcle for para recorrer por los elementos de un diccionario.

```
for nombre in tel:
  print(nombre)
```

```
William
John
Diego
```

```
for v in tel:
  print(v, tel[v])
```

```
William 22224466
John 22222222
Diego 22009999
```

```
for k, v in tel.items():
  print(k, v)
```

```
William 22224466
John 22222222
Diego 22009999
```

12.5.6 Intuición sobre matrices

Se le proporciona la siguiente tabla con algunos de los mejores videojuegos que han existido.

Game	Year	Genre
Bioshock	2007	First-person shooter
Breath of the Wild	2017	Action-adventure
Grand Theft Auto V	2013	Action-adventure
Minecraft	2011	Sandbox
Portal 2	2011	Puzzle-platformer
Resident Evil 4	2005	Survival Horror
The Last of Us	2013	Action-adventure

Usted se propone jugar cada título por semana, iniciando con el género *Acción-Aventura*:

Game	Year	Genre
Breath of the Wild	2017	Action-adventure
Grand Theft Auto V	2013	Action-adventure
The Last of Us	2013	Action-adventure

De la tabla, usted se da cuenta de que ya ha jugado *Breath of the Wild* previamente, por lo que decide reemplazarlo por otro título del mismo año y género. Busca la fila correspondiente y únicamente modifica el nombre.

Game	Year	Genre
Uncharted: The Lost Legacy	2017	Action-adventure
Grand Theft Auto V	2013	Action-adventure
The Last of Us	2013	Action-adventure

Por último, cuando llega a la semana 3, se da cuenta de que *The Last of Us* solo se encuentra disponible para PS3/PS4 y usted está jugando en PC. Así que decide cambiar por otro juego reciente que llamó su atención, en este caso, es necesario modificar la fila completa.

Game	Year	Genre
Uncharted: The Lost Legacy	2017	Action-adventure
Grand Theft Auto V	2013	Action-adventure
The Witcher 3: Wild Hunt	2015	Action role-playing

12.6 Matrices

Una matriz es una lista de listas, también se le conoce como lista 2D. Se puede acceder por listas o por elementos [lista][elemento].

12.6.1 Creación de Matrices

Se emplea la siguiente **sintaxis**.

```
nombreMatriz = [[elemento0, elemento1,...],[elemento0, elemento1],...]
```

Los elementos pueden ser de cualquier tipo y no necesariamente deben ser todos de la misma clase.

```
games = [["Bioshock", 2007, "First-person shooter"],
         ["Breath of the Wild", 2017, "Action-adventure"],
         ["Grand Theft Auto V", 2013, "Action-adventure"],
         ["Minecraft", 2011, "Sandbox"],
         ["Portal 2", 2011, "Puzzle-platformer"],
         ["Resident Evil 4", 2005, "Survival Horror"],
         ["The Last of Us", 2013, "Action-adventure"]]
print(games)
```

```
[['Bioshock', 2007, 'First-person shooter'], ['Breath of the Wild', 2017, 'Action-adventure'
```

12.6.2 Elementos de una matriz

Se trabajan de la misma forma que las listas. Considerar que ahora se trabaja con una lista dentro de una lista.

12.6.3 Número de elementos

Para obtener el número de listas, se utiliza el método len().

```
print("número de listas:",len(games)) #número de listas
```

```
número de listas: 7
```

12.6.4 Acceder o asignar elementos

Para acceder o asignar elementos, se debe indicar la posición utilizando [ ][ ] especificando la *lista* y *elemento*.

```
print(games[0]) #acceder a la primera lista
print(games[3][2]) #acceder a la cuarta lista, tercer elemento
```

```
['Bioshock', 2007, 'First-person shooter']
Sandbox
```

```
print(games[-1]) #desplegar la última lista
games[-1] = ["The Witcher 3: Wild Hunt", 2015, "Action role-playing"] #asignar nuevo valor a
print(games[-1]) #desplegar la última lista
```

```
['The Last of Us', 2013, 'Action-adventure']
['The Witcher 3: Wild Hunt', 2015, 'Action role-playing']
```

12.7 Recorrer elementos de una matriz

Para recorrer una matriz, se emplea la siguiente **sintaxis**:

```
for i in range(0, len(nombreMatriz)):
    bloque de código
```

```
for i in range(0, len(games)):
    print(games[i][0]) #desplegar los títulos de cada juego
```

```
Bioshock
Breath of the Wild
Grand Theft Auto V
Minecraft
Portal 2
Resident Evil 4
The Witcher 3: Wild Hunt
```

```
for i in range(0, len(games)):
    if(games[i][2]=="Action-adventure"): #si el género del juego es Acción-aventura
        print(games[i]) #desplegar el título del juego
```

```
['Breath of the Wild', 2017, 'Action-adventure']
['Grand Theft Auto V', 2013, 'Action-adventure']
```

13 Funciones

Bloque de código empleado para realizar una acción o retornar un valor.

13.0.1 e.g.

La función print realiza una **acción**. Despliega el string dentro del paréntesis (parámetro).

```
print("Bienvenido a mi programa")
```

```
Bienvenido a mi programa
```

13.0.2 e.g.

La función pow **retorna** un valor. Calcula la base (parámetro 1) elevada al exponente (parámetro 2).

$2^3 = 2 \cdot 2 \cdot 2 = 8$

```
import math
num = math.pow(2,3)
print(num)
```

```
8.0
```

13.1 Definición de funciones propias

Para crear una función propia se emplea la siguiente **sintaxis**.

```
def nombreFuncion(parámetro1, parámetro2, etc):
    bloque de código
    return expresión
```

El bloque de código de la función emplea indentación (tab). La utilización de parámetros y el retorno de valores es opcional.

13.1.1 Funciones que realizan una acción

13.1.2 e.g.

Se quiere evaluar si tres alumnos aprobaron el curso (nota mayor a 60) y desplegar el resultado.

Estudiante	Nota
John	80
Sarah	97
Michael	54

Para ello, se evalúa cada nota y se despliega cada resultado.

```
if(80>60):
    print("John aprobó el curso")
else:
    print("John reprobó el curso")

if(97>60):
    print("Sarah aprobó el curso")
else:
    print("Sarah reprobó el curso")

if(54>60):
    print("Michael aprobó el curso")
else:
    print("Michael reprobó el curso")
```

```
John aprobó el curso
Sarah aprobó el curso
Michael reprobó el curso
```

Sin embargo, este código es muy ineficiente debido a que es repetitivo. En este caso, resulta conveniente crear una función.

Se crea una función llamada *evaluar*, la cual emplea dos parámetros: *nombre* y *nota*.

```
def evaluar(nombre, nota):
    if(nota>60):
        print(nombre, "aprobó el curso")
    else:
        print(nombre, "reprobó el curso")
```

Ahora que se tiene una función, simplemente bastaría con llamarla y especificar los parámetros. La función realiza la acción de imprimir el resultado.

```
evaluar("John", 80)
evaluar("Sarah", 97)
evaluar("Michael", 54)
```

```
John aprobó el curso
Sarah aprobó el curso
Michael reprobó el curso
```

13.2 Funciones que retornan un valor

En ciertas ocasiones, se desea retornar un valor para seguirlo utilizando en el código en lugar de simplemente desplegar el resultado.

13.2.1 e. g.

Se quiere crear una función que calcule el precio en oferta de un producto. Al final se calcula el monto total de la compra.

Precio Original (Q)	Descuento (%)
100	25
35	60
70	50
10	5

```
def oferta(precioOriginal, descuento):
    precioActual = precioOriginal*(100-descuento)/100
    return precioActual

total = oferta(100,25) + oferta(35, 60) + oferta(70, 50) + oferta(10,5)
print("Total: Q",total)
```

```
Total: Q 133.5
```

14 Clases y objetos

Figura 14.1: ¿Quién es ese Pokémon?

14.0.1 Atributos (características)

1. Tipo
2. Altura
3. Peso
4. Nivel
5. HP (Hit Points)

14.0.2 Métodos (acciones)

1. Atacar (movimientos)
2. Defender (movimientos)
3. Aumentar de nivel
4. Evolucionar

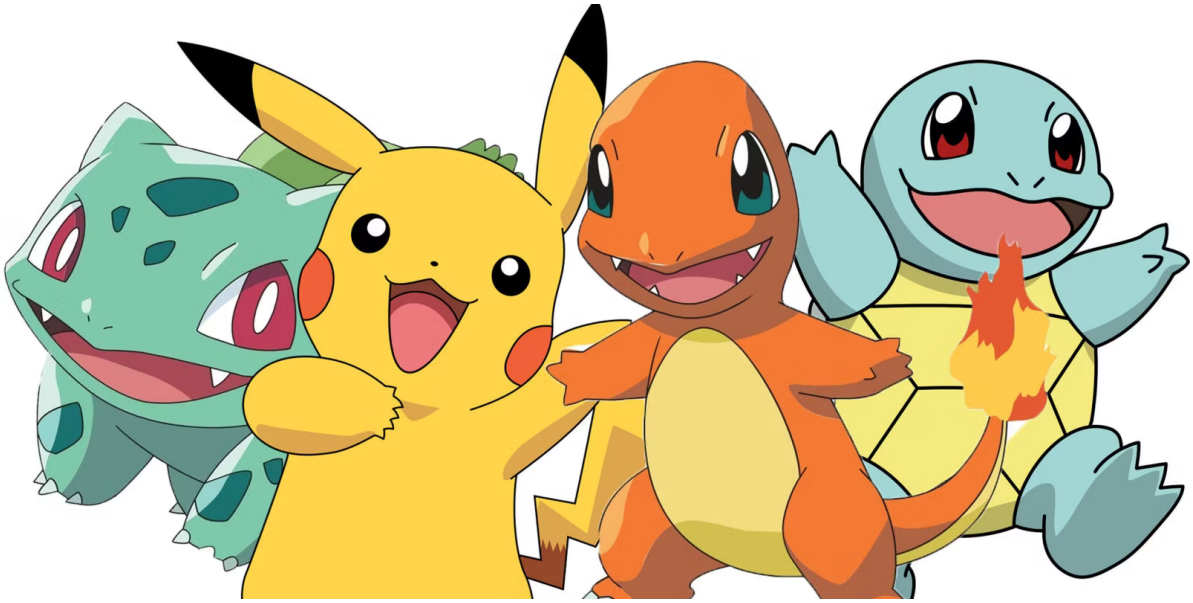

Figura 14.2: Pokémon

14.1 Clases

Base (o plano) para crear objetos. Compuesta por atributos y métodos.

14.1.1 Creación de Clases

Se emplea la siguiente **sintaxis**.

```
class NombreClase:
    def __init__(self, parámetro1, parámetro2, ...):
        self.atributo1 = parámetro1
        self.atributo2 = parámetro2
    def método1(self, parámetro1, parámetro2):
        bloque de código
```

```
class Pokemon:
    # constructor
    def __init__(self, tipo, nivel, ataque1, ataque2):
        # atributos
        self.tipo = tipo
        self.nivel = nivel
```

```
        self.ataque1 = ataque1
        self.ataque2 = ataque2
    # métodos = acciones
    def pokedex(self):
        print("-Tipo:",self.tipo)
        print("-Nivel:",self.nivel)
        print("-Ataque 1:",self.ataque1[0],"-Daño:",self.ataque1[1])
        print("-Ataque 2:",self.ataque2[0],"-Daño:",self.ataque2[1])
    def atacar1(self):
        print(f"¡Ha usado {self.ataque1[0]}!")
    def atacar2(self):
        print(f"¡Ha usado {self.ataque2[0]}!")
    def subirNivel(self):
        self.nivel += 1
        print(f"¡Ha subido al nivel {self.nivel}!")
```

14.2 Objetos

Instancia de una clase.

14.2.1 Crear un objeto

Se emplea la siguiente **sintaxis**.

```
nombreObjeto = nombreClase(atributo1, atributo2, ...)
```

```
#creación de un objeto de tipo Pokemon
charmander = Pokemon("Fuego", 1,["Arañazo", 6], ["Lanzallamas", 70])
```

```
charmander.pokedex() #llamar al método pokedex
```

```
-Tipo: Fuego
-Nivel: 1
-Ataque 1: Arañazo -Daño: 6
-Ataque 2: Lanzallamas -Daño: 70
```

```
charmander.atacar1() #llamar al método atacar1
```

```
¡Ha usado Arañazo!
```

```
charmander.subirNivel() #llamar al método subirNivel
```

```
¡Ha subido al nivel 2!
```

```
#creación de un segundo objeto de tipo Pokemon
pikachu = Pokemon("Eléctrico", 17,["Ataque Rápido", 8], ["Rayo", 80])
```

```
pikachu.pokedex()
```

```
-Tipo: Eléctrico
-Nivel: 17
-Ataque 1: Ataque Rápido -Daño: 8
-Ataque 2: Rayo -Daño: 80
```

```
pikachu.atacar1() #llamar al método atacar1
```

```
¡Ha usado Ataque Rápido!
```

```
pikachu.ataque1 = ["Voltio Cruel", 90] #asignar nuevo ataque
```

```
pikachu.atacar1() #llamar al método atacar1
```

```
¡Ha usado Voltio Cruel!
```

```
pikachu.subirNivel()
```

```
¡Ha subido al nivel 18!
```

```
pikachu.pokedex()
```

```
-Tipo: Eléctrico
-Nivel: 18
-Ataque 1: Voltio Cruel -Daño: 90
-Ataque 2: Rayo -Daño: 80
```