

Figura: Editor en línea OnlineGDB

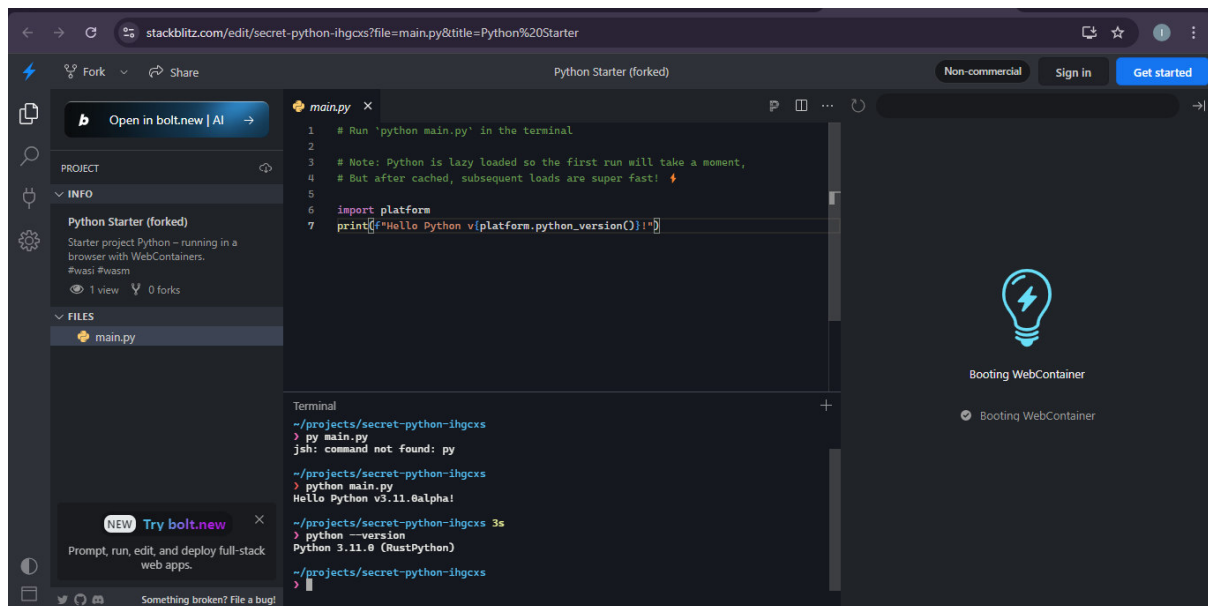


Figura: Editor en línea Stackblitz



**Nota:** Nosotros, y con el objeto de compartir el código y realizar prácticas, trabajaremos con [stackblitz](https://stackblitz.com) en la medida de lo posible.

## En Python ¿cómo se escribe el código?

### Sintaxis básica

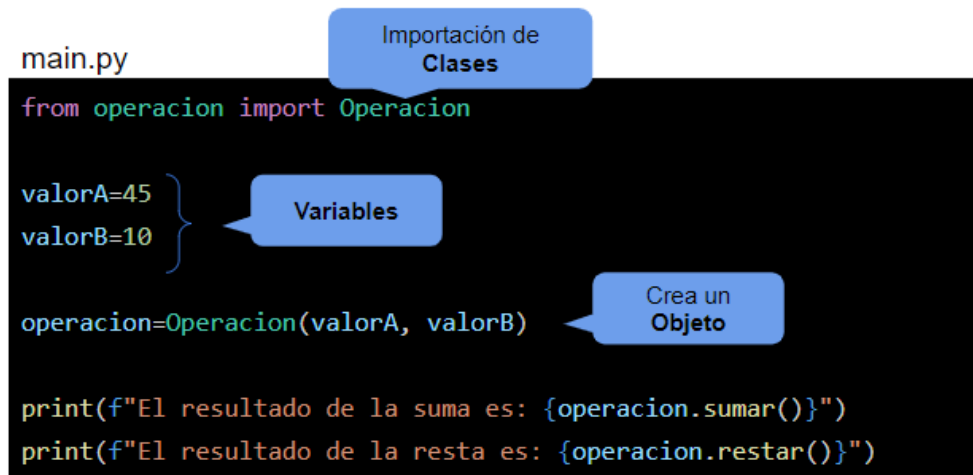
La sintaxis de un lenguaje de programación define una serie de reglas para escribir código. Cada lenguaje define su propia sintaxis.

En python, un programa está compuesto por:

- Módulos
- Funciones
- Clases
- Variables
- Declaraciones y expresiones

- Comentarios
- Entre otros

A continuación, compartimos unas imágenes resumen.



The image shows a code editor window titled 'main.py' with the following Python code:

```
from operacion import Operacion

valorA=45
valorB=10

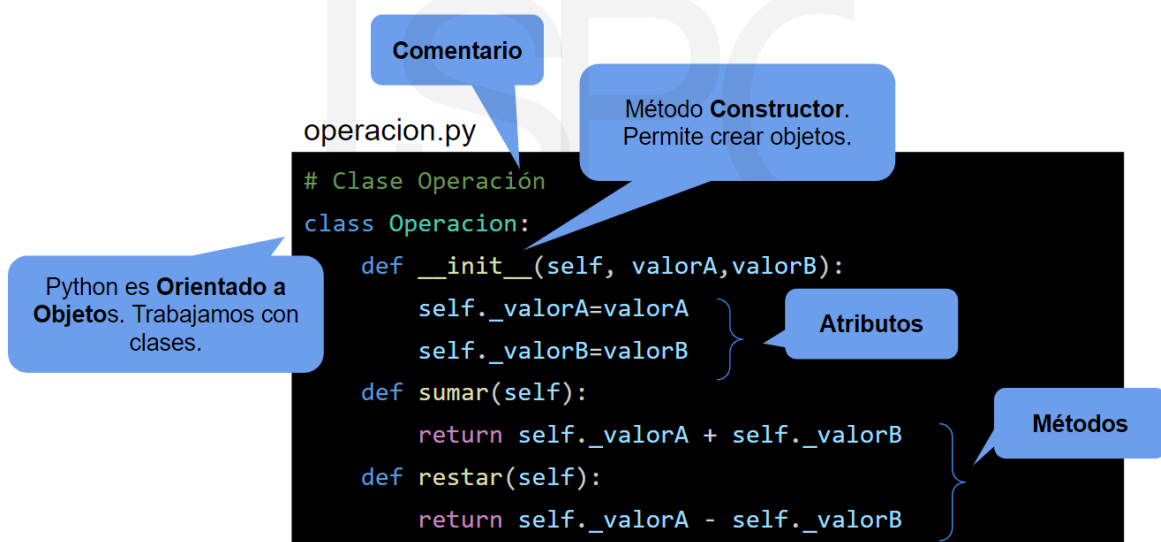
operacion=Operacion(valorA, valorB)

print(f"El resultado de la suma es: {operacion.sumar()}")
print(f"El resultado de la resta es: {operacion.restar()}")
```

Annotations (callouts) point to specific parts of the code:

- Importación de Clases**: Points to the line `from operacion import Operacion`.
- Variables**: Points to the lines `valorA=45` and `valorB=10`.
- Crea un Objeto**: Points to the line `operacion=Operacion(valorA, valorB)`.

Figura: Archivo main.py



The image shows a code editor window titled 'operacion.py' with the following Python code:

```
# Clase Operación
class Operacion:
    def __init__(self, valorA, valorB):
        self._valorA=valorA
        self._valorB=valorB
    def sumar(self):
        return self._valorA + self._valorB
    def restar(self):
        return self._valorA - self._valorB
```

Annotations (callouts) point to specific parts of the code:

- Comentario**: Points to the line `# Clase Operación`.
- Método Constructor. Permite crear objetos.**: Points to the `__init__` method.
- Atributos**: Points to the lines `self._valorA=valorA` and `self._valorB=valorB`.
- Métodos**: Points to the `sumar` and `restar` methods.
- Python es Orientado a Objetos. Trabajamos con clases.**: A general note on the left side of the code block.

Figura: Archivo operacion.py



**Nota:** No te preocupes si algunas palabras aún no comprendes, lo estaremos abordaremos en detalle más adelante.

## Identificadores

Los identificadores son los nombres que le damos a los elementos de un programa. Ej. variables, funciones, clases, etc.

Las reglas para los identificadores son:

- Los identificadores pueden contener letras y números. Sin embargo, un identificador no puede comenzar con un número.
- Los identificadores no pueden contener caracteres especiales ni espacios en blanco. A excepción del guión bajo (\_).
- Los identificadores no pueden ser palabras reservadas del lenguaje de programación.
- Los identificadores deben ser únicos.
- Los identificadores son case-sensitive.

El siguiente cuadro, muestra algunos ejemplos de identificadores válidos e inválidos

Identificadores válidos	Identificadores inválidos
miNombre	class
mi_nombre	mi nombre
num1	mi-nombre



**¡Investiga!** Es muy importante prestar atención a la nomenclatura o forma de escribir el nombre de una variable dado que por lo general estaremos trabajando con otros programadores ¿Conoces alguna nomenclatura para dar nombre a las variables?

Antes de continuar con la lectura, investiga las nomenclaturas recomendadas por la comunidad de Python.

## Elegir un buen nombre

Dar un buen nombre a un identificador en Python es esencial para la legibilidad y mantenibilidad del código fuente. Un nombre claro y descriptivo hace que el código fuente sea más fácil de entender y trabajar.

A continuación, se enumeran algunas recomendaciones:

- **Descriptivo:** El nombre de la variable debe describir claramente su propósito o contenido.
- **Notación snake\_case:** En Python, es común usar minúsculas y guiones bajos para separar palabras en los nombres de las variables.
- **Evitar Nombres Genéricos:** Nombres como x, temp, data, o var no son descriptivos y deben evitarse a menos que sea en contextos muy específicos (como en bucles cortos).
- **Conciso:** Aunque debe ser descriptivo, intenta mantener el nombre de la variable lo más corto posible sin perder claridad.

Como regla general se recomienda:

- Usar **sustantivos** para variables y nombres de clases.
- Usar **verbos** para funciones y métodos
- Usar **adjetivos** para variables booleanas.

## Palabras reservadas

Como todo lenguaje de programación tiene palabras que no deben usarse para nombrar nuestros objetos, variables, etc. python tienen sus palabras reservadas por lo que no podrás utilizarlas.

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while

async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

## Espacios en blanco y saltos de línea

En Python los espacios en blanco y saltos de línea son importantes para la legibilidad, la estructura del código y la indentación.

- **Espacios en blanco.** Se recomienda utilizar un espacio en blanco antes y después de los operadores (+, -, \*, etc.). Se debe además, evitar el uso excesivo de espacios en blanco porque puede causar problemas de indentación.
- **Indentación.** La indentación es obligatoria en Python y se utiliza para delinear los bloques de código. Se recomienda utilizar dos espacios en blanco en lugar de tabulaciones.
- **Saltos de línea.** Un salto de línea marca el fin de una línea de código y el inicio de otra. Se recomienda dejar una línea en blanco al final del archivo \*.py

Ejemplo de mala práctica:

```
a = "Leonel"
b = 20
c = [85, 90, 92]
```

Ejemplo de buena práctica:

```
first_name = "Leonel"
last_name = "Pérez"
birth_date = "1990-01-01"
```

## Comentarios

Los comentarios pueden ayudar a comprender el código. ¡Pero recuerda!

*"Si tú sientes que es necesario escribir un comentario,  
entonces es probable que tengas que refactorizar el código".*  
**Martin Fowler**

Esto significa que no debemos abusar de los comentarios en el código. **El código debería explicarse por sí mismo.**

Por ende, es recomendable usar los comentarios sólo para incluir información adicional como por ejemplo: el autor del código, sugerencias sobre una función/construcción, etc. El intérprete ignora los comentarios.

Python permite las siguientes formas para trabajar comentarios:

- **Comentarios de una sola línea:** Empiezan con el símbolo # y se extienden hasta el final de la línea.

Ejemplo:

```
def suma(a, b):  
    return a + b # Esto es un comentario de una sola línea
```

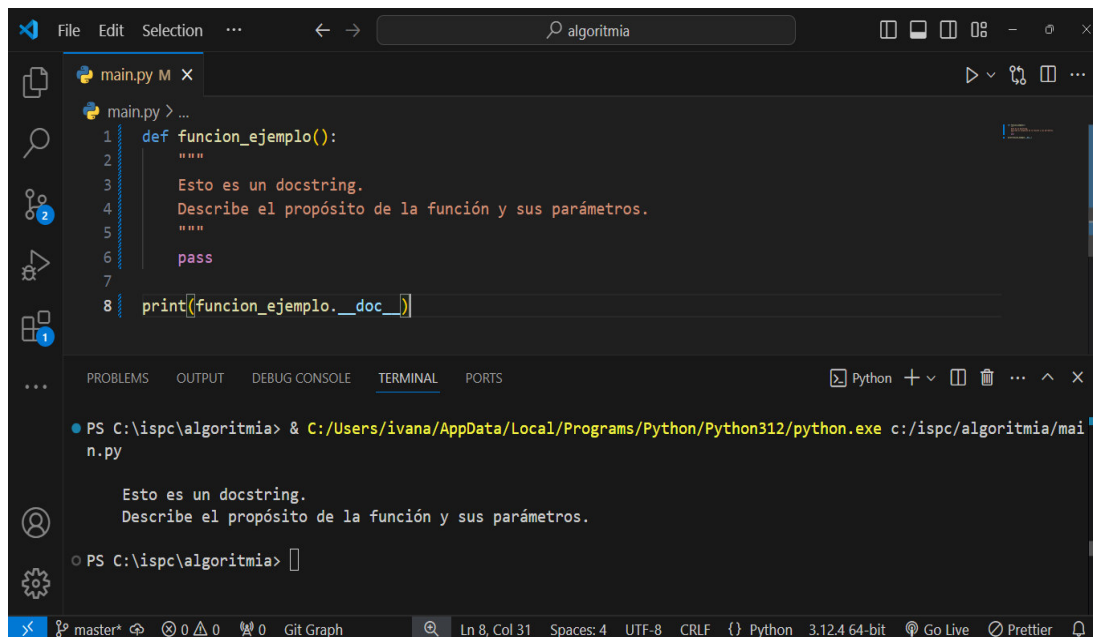
- **Comentarios de múltiples líneas:** no hay una sintaxis específica para comentarios de varias líneas, puedes usar varias líneas individuales de comentarios consecutivos o cadenas de texto de múltiples líneas no asignadas a ninguna variable.

Ejemplo:

```
# Esto es un comentario  
# que se extiende por  
# varias líneas  
  
"""  
También puedes usar triples comillas  
para comentarios de varias líneas,  
aunque esto es en realidad una cadena  
de texto no asignada a ninguna variable.  
"""
```

- **Docstrings:** Son comentarios que se utilizan para documentar módulos, clases y funciones. Utilizan triples comillas y son accesibles con la función help().

```
def funcion_ejemplo():  
    """  
    Esto es un docstring.  
    Describe el propósito de la función y sus parámetros.  
    """  
    pass  
  
print(funcion_ejemplo.__doc__)
```



```
main.py M X
main.py > ...
1 def funcion_ejemplo():
2     """
3     Esto es un docstring.
4     Describe el propósito de la función y sus parámetros.
5     """
6     pass
7
8 print(funcion_ejemplo.__doc__)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python + v [trash] ... ^ x

PS C:\ispc\algoritmia> & C:/Users/ivana/AppData/Local/Programs/Python/Python312/python.exe c:/ispc/algoritmia/main.py

Esto es un docstring.
Describe el propósito de la función y sus parámetros.

PS C:\ispc\algoritmia> [ ]

master* 0 0 0 Git Graph Ln 8, Col 31 Spaces: 4 UTF-8 CRLF Python 3.12.4 64-bit Go Live Prettier
```

Figura: Comentarios docsStrings

## Variables

Una variable por definición es un espacio de memoria que se utiliza para almacenar un valor durante el tiempo (scope) de ejecución del programa. La misma tiene asociado un identificador y un tipo de datos.

### Sintaxis

La sintaxis para declarar una variable en Python y asignarle un valor es muy simple dado que, sólo consiste en un identificador (nombre de la variable), el operador de asignación y el valor inicial a asignar (o guardar).

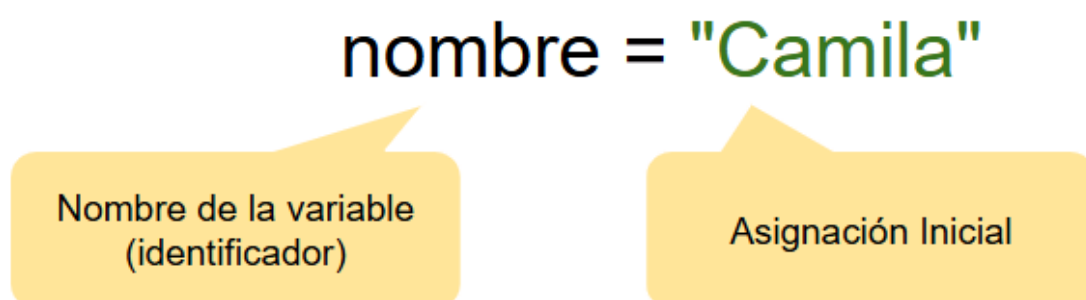


Figura: Sintaxis de Asignación de Variable

## Características de las variables

- **Asignación de Variables.** Las variables se crean al momento de asignarle un valor utilizando el operador de asignación “=”.

Ejemplo:

```
edad = 15 # edad es una variable de tipo entero  
nombre = "Maira" # nombre es una variable de tipo cadena de texto
```

- **Tipado dinámico.** No es necesario declarar el tipo de una variable al asignarle un valor. El tipo de variable se determina automáticamente según el valor asignado.

Ejemplo:

```
x = 10 # x es un entero  
x = "Hola" # x ahora es una cadena
```

- **Tipado fuerte.** No se admiten conversiones automáticas de tipos sin que se especifique explícitamente. Esto ayuda a prevenir errores que puedan surgir cuando se mezclan tipos de datos.

Ejemplo:

```
x = 10  
y = "5"  
resultado = x + y # Esto producirá un error TypeError
```

## Ámbito de Variable

Definimos ámbito de una variable a la región de código en la que dicha variable está definida y es accesible.



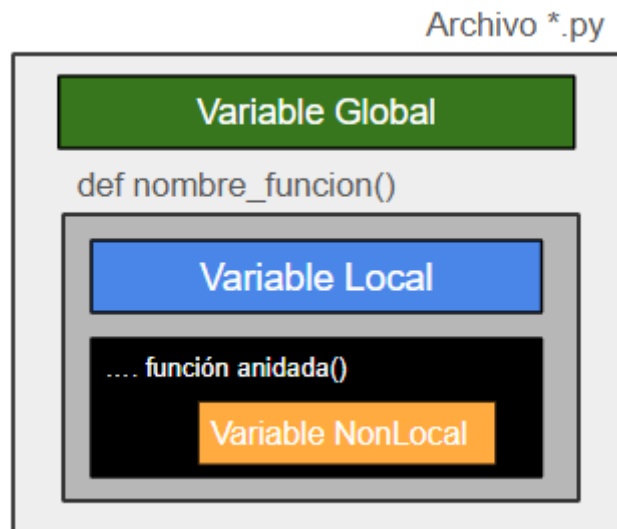


Figura: Ámbito de variables en Python

En Python existen los siguientes ámbitos de variables que enumeramos a continuación.

- **Ámbito global.** Las variables globales son definidas fuera de cualquier función y por ello, pueden ser accedidas desde cualquier parte del módulo (archivo), tanto dentro como fuera de la función o clase. Las mismas deben utilizarse con precaución para evitar conflictos.
- **Ámbito local.** Las variables locales son definidas dentro de una función (o método). Las mismas pueden ser accedidas sólo dentro de la función (o método). No afectan ni son afectadas por otras variables definidas fuera de la función.

Ejemplo:

```
mi_variable = "Global"

def mi_funcion():
    mi_variable = "Local"
    print(mi_variable) # Output: Local

#Ejemplo de uso
mi_funcion()
print(mi_variable) # Output: Global
```

- **Ámbito no local (Nonlocal).** Las variables no locales se utilizan en funciones anidadas. Las mismas, se definen en una función externa y son accesibles en las funciones internas. Para su declaración, se utiliza la palabra clave **nonlocal**.

Ejemplo:

```
def funcion_externa():  
    mi_variable = "Variable No Local"  
  
    def funcion_interna():  
        nonlocal mi_variable  
        mi_variable = "Variable Modificada"  
  
    print(mi_variable)    # Output: Variable No Local  
    funcion_interna()  
    print(mi_variable)    # Output: Variable Modificada  
  
#Ejemplo de uso  
funcion_externa()
```



**¡Investiga!.** Antes de continuar con la lectura investiga ¿Cuáles son los inconvenientes de utilizar variables globales? ¿En qué casos será conveniente utilizarlas?

## Conocer el valor y tipo de una variable

En Python, tenemos varias formas de conocer el valor de una variable y su tipo. Una de las formas consiste en utilizar las funciones integradas `print()` y `type()`.

### Usando la función `print()`

Para conocer el valor de una variable, podemos imprimir en consola utilizando la función `print()`.

Ejemplo:

```
nombre = "Camila"  
print(nombre) # Esto imprimirá: Camila
```

## Usando la función type()

Para conocer el tipo de una variable, podemos utilizar la función `type()`, que devuelve el tipo de datos.

Ejemplo:

```
edad = 20  
print(type(edad)) # Esto imprimirá: <class 'int'>
```

## Constantes

En Python, una constante es un valor que no cambia durante la ejecución del programa. A diferencia de las variables, las constantes están destinadas a permanecer inmutables una vez que se les asigna un valor.

En este punto es importante agregar que, aunque Python no tiene una sintaxis específica para definir constantes, es una convención usar nombres de variables en mayúsculas para indicar que son constantes y no deben ser modificadas.

## Características de las constantes

- **Inmutabilidad:** Una constante tiene un valor fijo que no debe cambiar durante la ejecución del programa.
- **Convención de Nomenclatura:** Se utilizan nombres en mayúsculas para las constantes para diferenciarlas de las variables. Por ejemplo, `PI = 3.1416`
- **Documentación:** Las constantes suelen documentarse claramente para indicar su propósito y el hecho de que no deben modificarse.

Ejemplo

```
PI = 3.14159 # Definición de constantes  
radio = 5  
circunferencia = 2 * PI * radio  
print(f"La circunferencia de un círculo con radio {radio} es {circunferencia}.")
```

## Tipos de datos

Los tipos de datos en Python especifican qué tipo de valor puede tener una variable y qué operaciones se pueden realizar con esos valores. A continuación listamos los principales tipos de datos en Python:

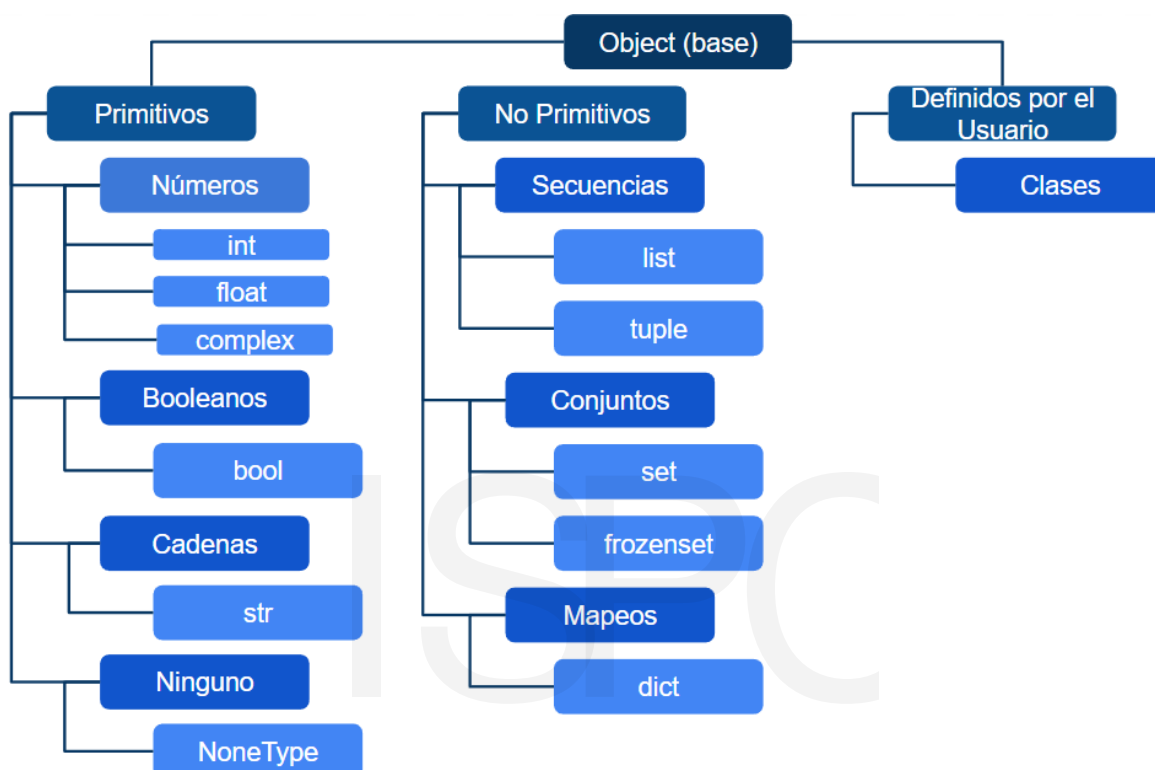


Figura: Tipos de datos principales de Python

### Tipos de Datos Primitivos

Los **tipos de datos primitivos** son los tipos básicos incorporados por el lenguaje de programación. La mayoría de los tipos de datos primitivos (a excepción de bytearray, no listado aquí) **son inmutables**. Esto quiere decir que no pueden modificarse una vez creados por lo que si cambias el valor de una variable, en realidad estás creando un nuevo objeto.

Ejemplo:

```

# Enteros
a = 5
a += 1 # Se crea un nuevo entero con valor 6

# Cadenas de texto

```

```
s = "Hola"
s += " mundo" # Se crea una nueva cadena con valor "Hola mundo"
```

La siguiente tabla ilustra los tipos primitivos enunciados arriba:

Tipo de Dato	Keyword	Descripción	Ejemplo
Numérico	int	Representa valores numéricos enteros.	<code>edad = 15</code>
Numérico	float	Representa valores numéricos flotantes (o decimal)	<code>pi = 3.14159</code>
Booleanos	bool	Representa un valores lógicos: True o False	<code>es_mayor_edad = True</code>
Cadentat	str	Representa una secuencia de caracteres Unicode	<code>nombre = "Maira"</code>
Ninguno	NoneType	Representa la ausencia de valor	<code>nada = None</code>

## *Tipos de Datos No Primitivos*

Los tipos de datos no primitivos en Python son estructuras de datos más complejas que están construidas a partir de los tipos primitivos.

La siguiente tabla ilustra los tipos de datos no primitivos listados arriba.

Tipo de Dato	Keyword	Descripción	Ejemplo
Listas	list	Representa una colección ordenada y mutable de elementos. Puedes cambiar, agregar o eliminar	<pre># Definición de una lista frutas = ["manzana", "naranja", "mandarina"]  # Acceso a elementos</pre>

		elementos después de crear la lista	<pre>print(frutas[0])      # Output: manzana  # Añadir un elemento frutas.append("uva") print(frutas)</pre>
Tuplas	tuple	Representan una colección ordenada e inmutable de elementos. Es decir que, una vez creadas, no puedes cambiar sus elementos.	<pre># Definición de una tupla punto = (10, 20)  # Acceso a elementos print(punto[0]) # Output: 10  # Intentar cambiar un elemento # (generará un error) try:     punto[0] = 30 except TypeError as e:     print(e)      # Output: 'tuple' object does not support item assignment</pre>
Conjuntos	set	Son colecciones desordenadas de elementos únicos. Los elementos duplicados no son permitidos	<pre># Definición de un conjunto colores = {"rojo", "verde", "azul"}  # Añadir un elemento colores.add("amarillo") print(colores)      # Output: {'amarillo', 'rojo', 'verde', 'azul'}  # Eliminar un elemento colores.remove("rojo") print(colores) # Output: {'amarillo', 'verde', 'azul'}</pre>
Conjuntos inmutables	frozenset	Son como los conjuntos (set), pero inmutables. Una vez creados, no puedes cambiar sus elementos.	<pre># Definición de un conjunto inmutable dias = frozenset(["lunes", "martes", "miércoles"])  # Intentar añadir un elemento # (generará un error) try:     dias.add("jueves") except AttributeError as e:     print(e)</pre>

Diccionarios	dict	Son colecciones de pares clave-valor. Cada clave debe ser única y se usa para acceder al valor asociado.	<pre># Definición de un diccionario persona = {"nombre": "Camila", "edad": 11}  # Acceso a un valor print(persona["nombre"]) # Output: Camila  # Añadir un nuevo par clave-valor persona["ciudad"] = "Buenos Aires" print(persona) # Output: {'nombre': 'Camila', 'edad': 11, 'ciudad': 'Buenos Aires'}</pre>
--------------	------	--	---

No te preocupes si algún ejemplo no comprendes hasta aquí. Estaremos abordando en este documento más adelante en detalle alguno de ellos.

## *Tipos de Datos Definidos por el Usuario*

Los tipos de datos definidos por el usuario son tipos de datos no primitivos que podemos modelar para crear estructuras de datos más complejas y específicas de acuerdo a las necesidades o requerimientos.

Tipo de Dato	Keyword	Descripción	Ejemplo
Enumera-ción	enum	Representa un conjunto de constantes con nombres y valores únicos.	<pre>from enum import Enum  class DiaSemana(Enum):     LUNES = 1     MARTES = 2     MIERCOLES = 3     JUEVES = 4     VIERNES = 5     SABADO = 6     DOMINGO = 7  # Uso de la enumeración dia = DiaSemana.MIERCOLES print(dia) # Output: DiaSemana.MIERCOLES print(dia.name) # Output: MIERCOLES print(dia.value) # Output: 3</pre>

Clases	class	Representa un objeto definido por el usuario.	<pre>class Rectangulo:     def __init__(self, ancho, alto):         self.ancho = ancho         self.alto = alto      def area(self):         return self.ancho * self.alto      def perimetro(self):         return 2 * (self.ancho + self.alto)  # Creando un objeto de la clase Rectangulo rectangulo1 = Rectangulo(5, 10) print(f"Área: {rectangulo1.area()}") # Output: Área: 50 print(f"Perímetro: {rectangulo1.perimetro()}") # Output: Perímetro: 30</pre>
--------	-------	---	---

No te preocupes si algún ejemplo no comprendes hasta aquí. Estaremos abordando POO (Programación Orientada a Objetos) en un documento posterior.

## Conversión entre tipos de datos

En Python, la convención de tipos está disponible mediante funciones que nos provee el propio lenguaje de programación.

A continuación se muestran algunos ejemplos sobre conversión de números entre distintos tipos.

```
# Conversión a entero
numero_convertido = int(45.6)
# Conversión a flotante
decimal_convertido = float(20)
# Conversión a complejo
complejo_convertido = complex(23)
```



El lenguaje nos provee más funciones para trabajar las conversiones e incluso se puede trabajar en distintos contextos, como conversiones para cálculos matemáticos, para operaciones con cadenas, etc.

## Operadores y Expresiones

Un operador define alguna función que se realizará con los datos. Los datos sobre los que trabajan los operadores se denominan operandos.

Considera la siguiente expresión:

$$3 + 5 = 8$$

Aquí los símbolos 3, 5 y 8 son **operandos** mientras que los símbolos + e = son **operadores**.

Python clasifica los operadores como sigue:

- Aritméticos
- De Comparación (o relacionales)
- Lógicos
- De Asignación
- Condicional Ternario

### Operadores Aritméticos

Asume que los valores de las variables A y B son 10 y 5 respectivamente.

Operador	Descripción	Ejemplo
+ (Adición)	Realiza la operación de suma y retorna la suma de los operandos.	A + B dará 15

- (Resta)	Realiza la operación de resta y retorna la diferencia de los operandos.	A - B dará 5
* (Multiplicación)	Realiza la operación de multiplicación y retorna el producto de los operandos	A * B dará 50
/ (División)	Realiza la operación de división y retorna el cociente	A / B dará 2
% (Módulo)	Realiza la operación de división y retorna el resto	A % B dará 0
** (Potenciación)	Realiza la operación de potenciación y retorna el resultado	A**B dará 100000

A continuación, puedes ver un ejemplo:

```
# Definición de variables
numero_1 = 10
numero_2 = 2
resultado = 0

# Suma
resultado = numero_1 + numero_2
print(f"Suma: {resultado}") # Output: Suma: 12

# Resta
resultado = numero_1 - numero_2
print(f"Resta:{resultado}") # Output: Resta: 8

# Multiplicación
resultado = numero_1 * numero_2
print(f"Multiplicación:{resultado}") # Output: Multiplicación: 20

# División
resultado = numero_1 / numero_2
print(f"División:{resultado}") # Output: División: Resultado: 5.0

# Resto
resultado = numero_1 % numero_2
print(f"Resto:{resultado}") # Output: Resto: 0
```



**¡Investiga!** Como puedes observar la función print utiliza las denominadas f-string. ¿Qué son? ¿Cómo se escriben? ¿Cuál es su utilidad? ¿Se puede escribir el mensaje a imprimir de otra forma? Si es así, ¿Cuál será la más óptima?

## Prioridad de los operadores aritméticos

Los operadores aritméticos tienen prioridad o precedencia que determina el orden en que se evalúan dentro de las expresiones.

Prioridad de operadores aritméticos

1. Potenciación (\*\*)
2. Multiplicación, División, División Entera y Módulo (\*, /, //, %)
3. Adición y Sustracción (+, -)

Para comprenderlo mejor veamos el siguiente ejemplo:

```
resultado = 20 + 3 * 2 ** 2 - 8 / 4 % 3 + 10  
print(resultado) # output: 40.0
```

En base a la prioridad antes mencionada:

1. Se evalúan los paréntesis (no tenemos ninguno en este caso)
2. Se calculan las potencias ( $2^{**}2=4$ ). Entonces la expresión resultante es:  $20 + 3 * 4 - 8 / 4 \% 3 + 10$
3. Se calculan las operaciones de multiplicación y división ( $3*4 =12$  y  $8/4=2$ ). Entonces la expresión resultante es:  $20 + 12 - 2 \% 3 + 10$
4. Se calcula modulo ( $2\%3=2$ ). Entonces la expresión resultante es:  $20 + 12 - 2 + 10$
5. Finalmente las sumas y restas. El resultado: 40

## Operadores de Comparación (o relacionales)

Los operadores relacionales comparan dos entidades y devuelven un valor booleano, es decir, verdadero/falso.

Asume que el valor de A es 10 y B es 20.

Operador		Descripción	Ejemplo
==	Igualdad	Devuelve verdadero (true) si ambos operandos son iguales.	(A==B) es falso
!=	Desigualdad	Devuelve verdadero (true) si ambos operandos no son iguales.	(A != B) es verdadero.
>=	Mayor que	Devuelve verdadero (true) si el operando de la izquierda es mayor o igual que el operando de la derecha.	(A >= B) es falso
>	Mayor	Devuelve verdadero (true) si el operando de la izquierda es mayor que el operando de la derecha.	(A > B) es falso
<	Menor	Devuelve verdadero (true) si el operando de la izquierda es menor que el operando de la derecha.	(A < B) es verdadero
<=	Menor que	Devuelve verdadero (true) si el operando de la izquierda es menor o igual que el operando de la derecha	(A <= B) es verdadero

A continuación, puedes ver un ejemplo:

```

numero_1 = 5
numero_2 = 9
print("Valor de número 1:", numero_1)
print("Valor de número 2:", numero_2)
resultado = numero_1 > numero_2
print(f"{numero_1} es más grande que {numero_2}?:", resultado)
resultado = numero_1 < numero_2
print(f"{numero_1} es menor que {numero_2}?:", resultado)
resultado = numero_1 >= numero_2
print(f"{numero_1} es mayor o igual que {numero_2}?:", resultado)
resultado = numero_1 <= numero_2
print(f"{numero_1} es menor o igual que {numero_2}?:", resultado)
resultado = numero_1 == numero_2
print(f"{numero_1} es igual que {numero_2}?:", resultado)
resultado = numero_1 != numero_2
print(f"{numero_1} es diferente a {numero_2}?:", resultado)

```

```

"""Output:
Valor de número 1: 5
Valor de número 2: 9
5 es más grande que 9?: False
5 es menor que 9?: True
5 es mayor o igual que 9?: False
5 es menor o igual que 9?: True
5 es igual que 9?: False
5 es diferente a 9?: True
"""

```

## Operadores Lógicos

Los operadores lógicos se utilizan para combinar dos o más condiciones y retornan también un valor booleano: true o false.

Asume que A es 10 y B es 20.

Operador	Descripción	Ejemplo
and (lógico Y)	El operador AND retorna true sólo si todas las condiciones son true	(A > 10 and B > 10) es falso
or (lógico O)	El operador OR retorna true si al menos una de las condiciones es true	(A > 10 or B > 10) es verdadero
not (lógico no)	El operador no (denominado NOT) retorna el valor inverso de la expresión resultante.	not (A > 10) es verdadero

A continuación puedes ver un ejemplo:

```

numero_1 = 20
numero_2 = 90

print("Valor del a:", numero_1, ", Valor del b:", numero_2)

resultado = ((numero_1 > 50) and (numero_2 > 80))
print("(a>50) and (b>80):", resultado)

resultado = ((numero_1 > 50) or (numero_2 > 80))
print("(a>50) or (b>80):", resultado)

resultado = not ((numero_1 > 50) and (numero_2 > 80))
print("not ((a>50) and (b>80)):", resultado)

```

```

"""Output:
Valor del a: 20 , Valor del b: 90
(a>50) and (b>80): False
(a>50) or (b>80): True
not ((a>50) and (b>80)): True
"""

```

## Operadores de Asignación

Los operadores de asignación se utilizan para asignar valores a las variables.

Operator	Description	Example
= (Asignación Simple)	Asigna valores del operando del lado derecho al operando del lado izquierdo	C = A + B asignará el valor de A + B a C
+= (Agrega y Asigna)	Suma el operando derecho al operando izquierdo y asigna el resultado al operando izquierdo.	C += A es equivalente a C = C + A
-= (Resta y Asigna)	Resta el operando derecho del operando izquierdo y asigna el resultado al operando izquierdo.	C -= A es equivalente a C = C - A
*= (Multiplica y Asigna)	Multiplica el operando derecho con el operando izquierdo y asigna el resultado al operando izquierdo.	C *= A es equivalente a C = C * A
/= (Divide y Asigna)	Divide el operando izquierdo con el operando derecho y asigna el resultado al operando izquierdo.	C /= A es equivalente a C = C / A

A continuación puedes ver un ejemplo:

```

numero_1 = 12
numero_2 = 10

resultado=numero_1
resultado = numero_2
print(f"{numero_1} = {numero_2}=>", resultado)

```

```

resultado=numero_1
resultado += numero_2
print(f"{numero_1} += {numero_2}=>", resultado)

resultado=numero_1
resultado -= numero_2
print(f"{numero_1} -= {numero_2}=>", resultado)

resultado=numero_1
resultado *= numero_2
print(f"{numero_1} *= {numero_2}=>", resultado)

resultado=numero_1
resultado /= numero_2
print(f"{numero_1} /= {numero_2}=>", resultado)

resultado=numero_1
resultado %= numero_2
print(f"{numero_1} %= {numero_2}=>", resultado)

"""Output:
12 = 10=> 10
12 += 10=> 22
12 -= 10=> 2
12 *= 10=> 120
12 /= 10=> 1.2
12 %= 10=> 2
"""

```

## Operador condicional (ternario)

El operador ternario, conocido también como operador condicional permite evaluar una condición en una sola línea de código y devolver un valor basado en si la condición es verdadera o falsa.:

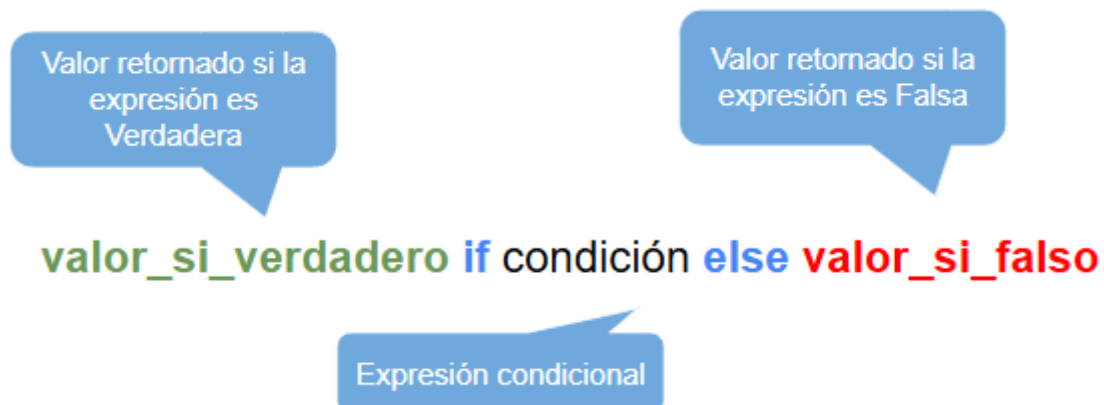


Figura: Operador ternario

Si la condición es verdadera (true), el operador retorna el **valor\_si\_verdadero**, de lo contrario retorna el **valor\_si\_falso**. Puedes usar el operador condicional en cualquier lugar que dónde uses un operador estándar.

Por ejemplo,

```
# Asignar valores a variables
numero_1 = 30
numero_2 = 20

# Usar el operador ternario para encontrar el valor mayor
mayor = numero_1 if numero_1 > numero_2 else numero_2
print(f"El mayor valor es: {mayor}") #Output: El mayor valor es: 30
```

En este ejemplo, la expresión evalúa si `numero_1` es mayor que `numero_2`. Si la condición es verdadera, devuelve `numero_1`; de lo contrario, devuelve `numero_2`.



### ¡Pongámonos a prueba!

1. Escribe un programa que pida al usuario el radio de un círculo y calcule el área.
2. Escribe un programa que convierta una temperatura dada en grados Celsius a grados Fahrenheit.
3. Escribe un programa que pida al usuario los dos catetos de un triángulo rectángulo y calcule la hipotenusa.
4. Escribe un programa que pida al usuario su año de nacimiento, calcule su edad y genere un mensaje de saludo personalizado que incluya su nombre y la edad calculada.

## ¿Por dónde empiezo?

Si no sabes por dónde comenzar para resolver estos ejercicios intenta con las fases de resolución de problemas que resumido podemos observar.



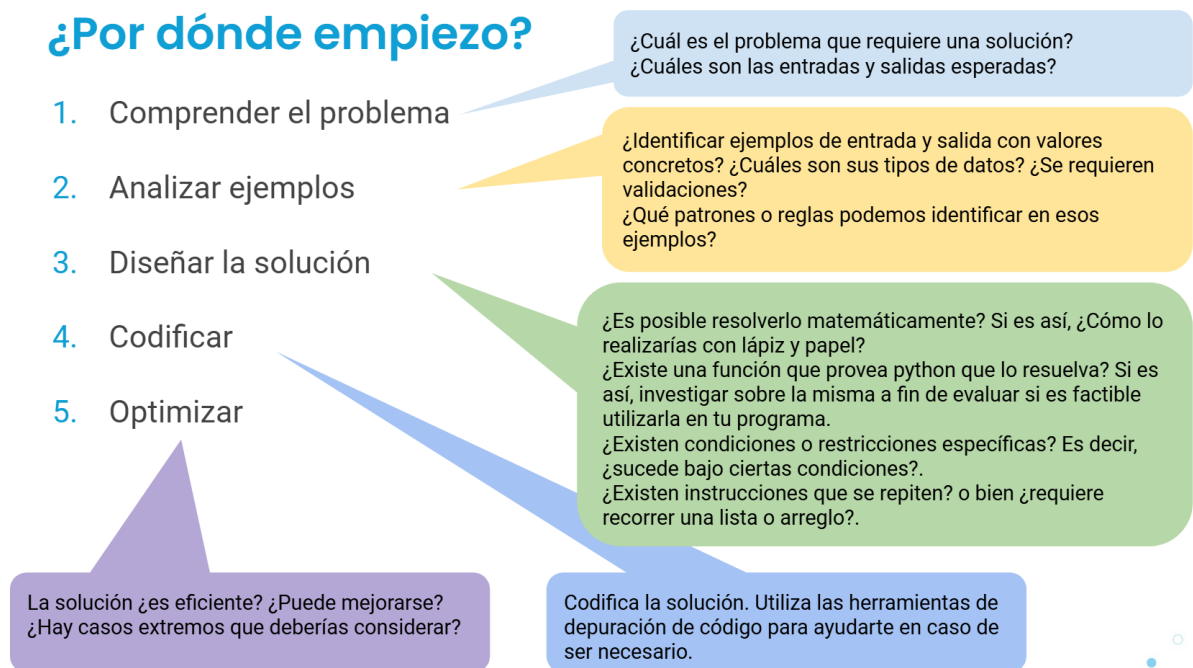


Figura: Fases para la resolución de problemas

Por ejemplo, para resolver el ejercicio “Escribe un programa que pida al usuario el radio de un círculo y calcule el área.”

1- **Comprender el problema.** Para ello, podrías hacerte preguntas tales como ¿Qué es el área? ¿Cómo se calcula el área de un círculo? Investiga y encuentra la fórmula.

2- **Analizar ejemplo.** Utiliza lápiz y papel. Identifica las entradas y salidas con valores concretos. Ej. para un radio es igual 2 el área de un círculo es igual a 12.56.

3- **Diseñar la solución.** Resuelve mediante alguna técnica de diseño si es posible. Investiga si no existe una función de Python que lo resuelva. Identifica posibles restricciones que debas tener en cuenta, ej. división por cero, etc.

4- **Codificar.** Escribe el código y evalúa que funcione correctamente para todos pares de entrada/salida.

5- **Optimizar.** Si el código funciona, investiga si esa solución puede mejorarse en términos de código limpio, performance, etc.

## ¿Qué significa depurar el código? ¿Cómo lo hago?

Depurar el código en programación se refiere al proceso de encontrar y corregir errores o bugs en un programa de computadora.

¿Por qué depuramos?

- El código que escribimos no hace lo que esperamos.

La depuración de código nos permite:

- Identificar y resolver problemas en el código.
- Encontrar el punto exacto dónde se cometió el error de programación.
- Realizar correcciones para que el programa pueda continuar ejecutándose.

## *Pasos para realizar la depuración de código con VSCode*

1- **Aclarar el problema.** Es importante que antes nos hagamos preguntas tales como:

- ¿Qué se espera que haga el código?
- ¿Qué pasó en su lugar?
- ¿Se produjo una excepción (error) durante la ejecución? Si es así, investiga posibles soluciones. (Tip, buena fuente <https://es.stackoverflow.com/> )

2- **Examinar suposiciones.** Cuestiona las suposiciones que te llevaron a esperar ese resultado. (Tip. Revisa la fuente oficial).

3- **Utilizar el depurador de VSCode** para ejecutar línea a línea e identificar qué está sucediendo.

Para ello:

- a. Establecer un punto de interrupción haciendo click en la línea de código dónde se desea que se detenga la ejecución del programa.

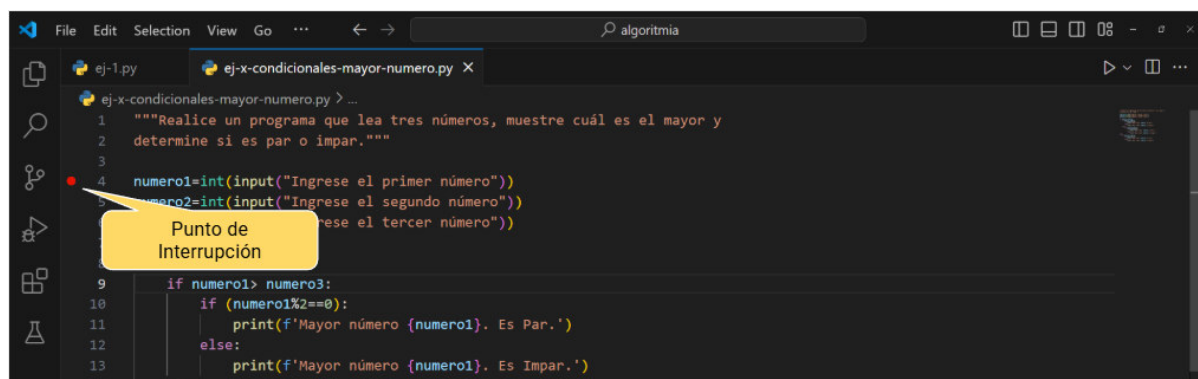


Figura: Punto de Interrupción

- b. A continuación, hacer click en la herramienta "Run and debug" o bien, Ctrl+Shift+D

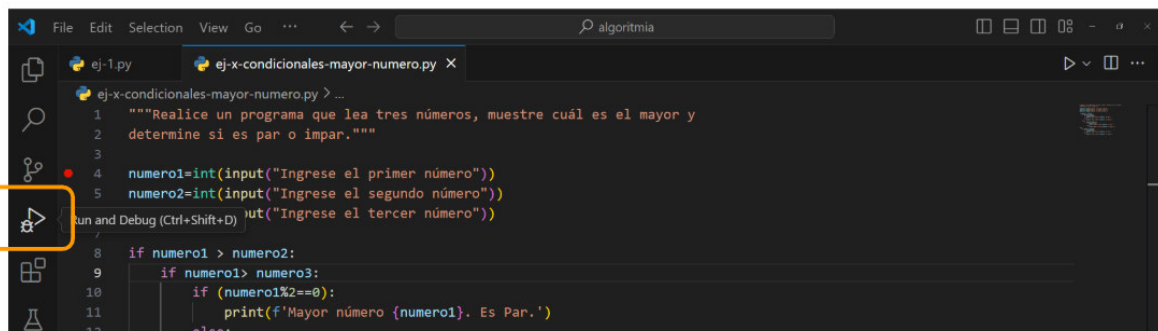


Figura: Icono Run and Debug

- c. Se abrirá la ventana Run and Debug, hacer click en el botón “Run and Debug”.

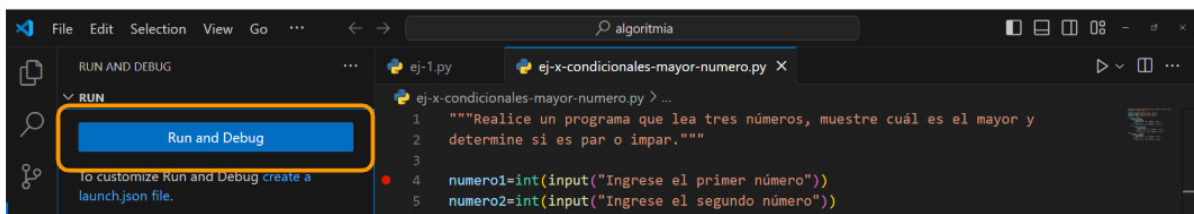


Figura: Botón Run and Debug

- d. Luego, seleccionar la opción Python Debugger como sigue:

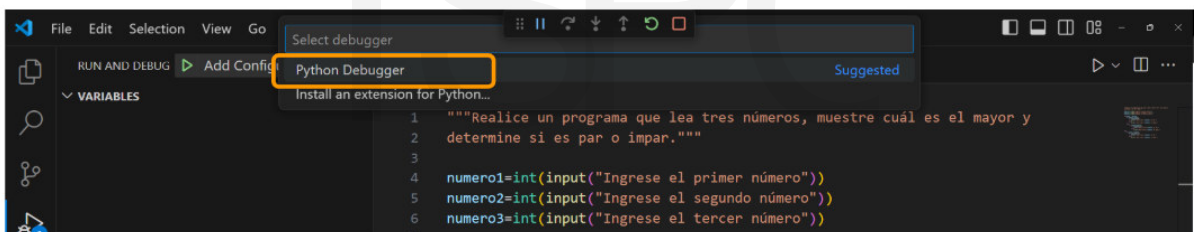


Figura: Selección del Debugger

- e. Si todo va bien, se podrá observar en la parte superior de la ventana las herramientas del debugger. El programa parará en la línea dónde especificaste el punto de interrupción y después podrás ejecutar línea a línea utilizando las herramientas o bien presionando F11.

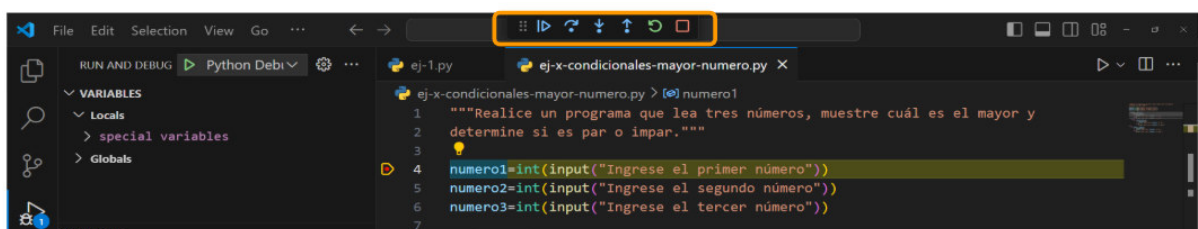


Figura: Herramientas del debugger

- f. Finalmente, observa que en la Run and Debug se pueden ver los valores de las variables en tiempo de ejecución.

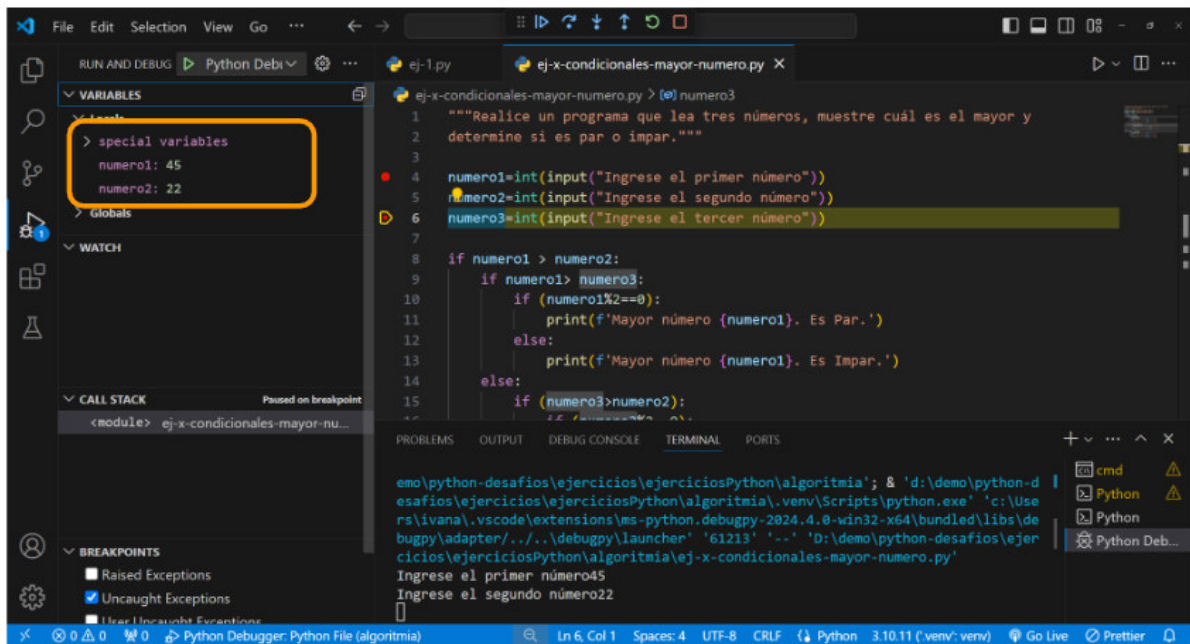


Figura: Visualización de variables en tiempo de ejecución

Podrás ver los valores que van tomando las variables, si el programa ejecuta o no el bloque contenido dentro de una estructura condicional o repetitiva, etc.

## Estructuras de Control

Las estructuras de control, permiten dirigir el flujo de ejecución de un programa, tomar decisiones, repetir acciones y gestionar la lógica de manera eficiente. Si ellas, los programas serían simplemente una serie de instrucciones lineales como hemos observado hasta ahora.

Existen diferentes tipos de estructuras de control: estructuras secuenciales, condicionales (o de selección) y repetitivas (o bucles).