1. Repaso sobre ámbitos de una variable

Como se destacó en la clase anterior, una variable es solo visible y existe únicamente dentro de su ámbito. Además, ésta existe mientras el ámbito esté activo y desaparecen al cerrar el mismo.

Programa 1: Variables locales y globales

```
1  ''' variable global '''
2  # Se destruye al finalizar la ejecución del script:
3  variable_G = 100
4
5  def main(): # Ámbito local de main()
6    variable_A = 0 # se crea la variable_A
7
8    if variable_A == 0: # Ámbito del if
9      variable_B = 1    # se crea la variabla_B
10     print(f"{variable_A}, {variable_B}, {variable_G}")
11     # se "destruye" la variable_B al salir del if
12
13   # se destruye la variable_A
14
15 main()
```

En la porción de código anterior podemos observar que dentro del ámbito local de `main()` se crea y puede acceder a la `variable_A`, dentro del mismo se crea el ámbito de una sentencia de control `if` en la *linea 8* donde se instancia la variable `variable_B`. Como este último está dentro del ámbito del `main()` puede acceder tanto a la `variable_A` como a la `variable_B`, pero esta última es solo accesible desde dentro del if y deja de existir en la *línea 11* finaliza el bloque. La `variable_G`, al ser declarada fuera de todo ámbito se considera global y accesible desde cualquier parte del archivo.

Este concepto es importante para "ahorrar" memoria al sistema operativo, como las variables globales existen hasta que se termine el programa es deseable evitarlas y "particionar" nuestro programa en funciones que posean sus propias variables locales que utilicen la memoria mientras se ejecutan y la liberen al finalizar su ejecución.

Programa 2: Localidad en funciones

```
1  def pedir_entero(msj): # se crea 'msj'
2    while True:
3      try:
4        n = int(input(msj)) # se crea 'n'
5        return n            # se devuelve el valor de 'n'
6      except ValueError:
7        print("Error de ingreso, intente nuevamente...")
8
9    # cuando la función retorna, se elimina 'msj'
10
11 def sumar(x, y): # se crea 'x' e 'y'
12   return x + y # se retorna el resultado de 'x'+'y'
13   # se destruyen 'x' e 'y'
14
```

```
15 def main():
16   a = pedir_entero("Ingrese un entero: ") # se crea 'a'
17   b = pedir_entero("Ingrese un entero: ") # se crea 'b'
18   print(f"{a}+{b}={sumar(a,b)}")
19   # se destruyen 'a' y 'b'
20
21 main()
```

1.1. Inmutabilidad

En Python existen dos tipos de datos, denominados:

mutables: admiten el uso de la asignación (=) para cambiar el valor de alguno de sus estados/valores.

inmutable: No permiten el uso de la asignación (=) para cambiar el valor de alguno de sus estados/valores.

La mayoría de los tipos de datos que se utilizaron hasta el momento tienen la característica de ser inmutables. Es decir, no podemos cambiar el valor que almacenan. Analicemos el siguiente ejemplo en el **REPL**:

```
>>> hex(id(10))  # id del objeto '10'
'0x7fd82ff4c210'
>>> a = 10
>>> hex(id(a))  # el id del '10' guardado en 'a'
'0x7fd82ff4c210'
>>> a = a * 2   # se crea un nuevo valor, generado por el
   inmutable 10 multiplicado x2
>>> hex(id(a))  # el id del nuevo valor guardado en 'a'
'0x7fd82ff4c350'
>>> hex(id(20)) # id del objeto '20' (ahora en 'a')
'0x7fd82ff4c350'
>>>
```

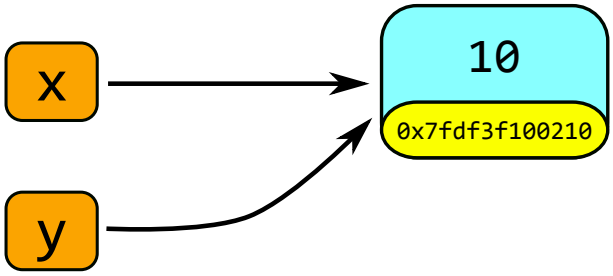
En Python podemos saber la referencia única a un objeto en la memoria a través de la función `id()`. Gracias a ella, podemos observar que en realidad la variable `'a'` almacena un objeto del tipo `int` inmutable (`10`), y que al utilizar el operador asignación (=), en realidad, se está reasignando a la variable un nuevo objeto inmutable (`20`).

Los objeto inmutables en Python son los datos numéricos (enteros y flotantes), los *strings* y las tuplas (que todavía no se han utilizado). Estos no admiten cambios, y darán error al intentar modificarlos:

```
>>> s = "abc"
>>> s[1] = "B" # Se intenta modifcar la 'b' dentro del string
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

2. Referencias

Si bien no es una terminología habitual en Python, lo cierto es que todas las variables "son referencias". Es decir que lo almacenado en una variable es la ubicación de un objeto en la memoria. Podemos afirmar que en el siguiente ejemplo x e y "apuntan" al mismo dato almacenado en la memoria.



Programa 3: Referencia a una variable

```
1  def main():
2    x = 10 # se crea 'x' con referencia al objeto int '10'
3    y = x  # se crea 'y' con referncia al objeto int '10'
4
5    print(f"id(x)={hex(id(x))}, id(y)={hex(id(y))}")
6
7  main()
```

Salida del programa:

```
id(x)=0x7fdf3f100210, id(y)=0x7fdf3f100210
```

La variable **'y'** y **'x'** hacen referencia al mismo objeto en la memoria. Pero como ya vimos anteriormente, los números en Python son **inmutables**. Eso quiere decir que por más que pasemos la referencia, estos no pueden ser modificados.

2.1. Parámetros e inmutabilidad

Como se explicó, las variables contienen la referencia a una variable, y al asignarlos con el operador **'='** estos pierden la referencia original y pasan a *apuntar* a otros objeto en la memoria.

Analice el siguiente *script*:

```
1  def incrementar(x):
2    x = x + 1
3
4  def main():
5    x = 10
6    print(f"Antes de incrementar() -> x = {x}")
7    incrementar(x)
8    print(f"Después de incrementar() -> x = {x}")
9
10 main()
```

Al leer este programa se podría interpretar que mostrará inicialmente un 10 y luego un 11, pero el verdadero resultado es:

```
Antes de incrementar() -> x = 10
Después de incrementar() -> x = 10
```

¿Qué ocurre? Para diferenciar ambas variables con el mismo nombre diremos `main.x` e `incrementar.x` respectivamente. Como se observa al invocar o llamar a la función `incrementar()` le pasamos el objeto `main.x`. En ese momento ambos apuntan al mismo objeto en memoria. Sin embargo, dentro del ámbito de la función `incrementar.x` pasa a apuntar a otro objeto en el instante en que se le suma otro valor en la *línea 2*. Esto es debido a la inmutabilidad propia de los `int` en Python.

3. Especificación de tipos

En versiones reciente de Python, se agregó la posibilidad de indicarle a otros programadores qué tipo de datos se espera como parámetro de una función y qué tipo de valor devuelven. Si bien, como ya hemos analizado, Python tiene *tipado dinámico*, es una pequeña ayuda para evitar conflictos no deseados.

```
1  ''' función que recibe un entero y devuelve un entero '''
2  def incrementar(x:int) -> int:
3      return x + 1
4
5  def main():
6      x = 10
7      print(f"Antes de incrementar() -> x = {x}")
8      x = incrementar(x)
9      print(f"Después de incrementar() -> x = {x}")
10
11 main()
```

Como se observa es posible indicar a través del carácter `':'` el tipo de dato esperado luego de nombrar la variable en cuestión. Además, si se trata de una función, podemos especificar con `'->'` el tipo de dato que retorna. Esto no quiere decir que podamos enviar un `float`, este código seguirá funcionando. Pero hay software de auditoría para nuestros *scripts* de Python que nos advertirán si no estamos utilizando la o las funciones como estaban pensadas desde su diseño.

En el ejemplo anterior, no hay problema, ya que es posible sumar 1 tanto en enteros como flotantes. Pero el siguiente caso es distinto:

```
1  def concat(s1:str, s2:str) -> str:
2      return s1 + s2
3
4  def main():
5      c = concat("hola ", 1)
6      print(f"{c}")
7
8  main()
```

Esto nos arrojará un claro `TypeError` porque Python no sabe como sumar un `str` con un `int`. Dejar que el error se produzca es un método válido de funcionalidad. Pero se podría hacer una función más robusta de la siguiente manera:

```
def concat(s1:str, s2:str) -> str:
    return str(s1) + str(s2)
```

Es una buena práctica especificar los tipos de las variables en las funciones y si estas devuelven algún valor también, **siempre y cuando el algoritmo solo funcione con esos tipos de datos específicos**. Es decir, que si el tipo de dato es importante para el algoritmo, debería aconsejarle al programador que utilizará la función, qué tipo de datos son los necesarios para que el algoritmo se comporte correctamente.