Manual introductorio a Python
v1

Kenny J. Tinoco

Febrero de 2025

Índice

1. Programación imperativa **1**

1.1. Variables 1

1.1.1. Listas 3

1.2. Operadores 5

1.3. Números 6

1.4. Sentencias de control 7

1.5. Cíclos 9

1.5.1. While 9

1.5.2. For 10

1.5.3. Break y Continue 10

1.6. Funciones 11

1.7. Ejercicios 12

2. Programación orientada a objetos **14**

---

1. Programación imperativa

1.1. Variables

Una **variable** en Python es el nombre que se asigna un dato almacenado con los cuales se harán operaciones, estos datos pueden ser de distintas naturalezas que denotaremos como **tipo**, es decir, pueden ser de tipos enteros (**int**), decimales (**float**), textos (**str**), valores booleanos (**bool**), listas (**list**), etc.

Para crear una variable escribimos su nombre y luego asignamos su valor por medio del operador de asignación "=".

Una cosa importante a tener en cuenta es que el nombre de una variable tiene ciertas restricciones, la primera es que el nombre no puede iniciar por un número, la segunda es que no puede ser una palabra reservada del lenguaje y la tercera es que no puede tener caracteres especiales excepto el "_", vamos algunos ejemplos.

```
1   string = "Un string"
2   anotherStr = 'a'
3   integer = 4
4   x = -1
5   boolean = x > 0  # (es falso)
6   isTrue = True
7   isFalse = False
```

Variables válidas.

```
1   a1239892874 = 1
2   x10 = 2
3   AGE = 67
4   my_lastname = "lopez"
5   _name = "luis"
```

Variables inválidas.

```
1   10x = 1
2   name-value = "luis"
3   first% = 3
```

Otro aspecto muy importante es escribir nombre de variables que tenga un sentido dentro del código, evitar escribir variables de un solo carácter y en su lugar escribir un sustantivo.

Vamos a entender como **sentencia** a cualquier operación sobre un valor, por ejemplo el siguiente código tiene dos sentencias.

```
1   mySurname = "Tinoco"
2   print(mySurname)
```

Todo programa en Python está formado por un conjunto de sentencias que se escriben con el fin de obtener un resultado de interés. Cada sentencia se escribe en una línea, sin embargo, es posible escribir más de una sentencia en una sola línea de código separándolas por punto y coma.

```
1   mySurname = "Tinoco"; print(mySurname)
```

Se recomienda nunca escribir más de una sentencia en una línea.

Si necesitamos escribir texto como una manera de señalar algún aspecto del código, podemos usar el símbolo "#" la cual le indica a lenguaje que ignore todo lo que despues del símbolo, es decir, un comentario.

```
# Este es un comentario y no se ejecuta.

aString = "Un texto" #Este es otro comentario
```

Los comentarios si bien son parte del lenguaje, en principio no deben usar casi nunca, ya que el código debe ser lo suficientemente claro para tener que comentarlo. En resumen NUNCA USAR COMENTARIOS.

La **identación** en líneas de código tiene una gran importancia ya que Python forma bloques de código por medio de la identación, toma relevancia el tema de funciones, bloques condicionales, sentencias de control y clases.

```
mySurname = "Tinoco"
    print(mySurname)
```

El código anterior falla debido al significado especial de la identación.

Cuando hacemos un código como el siguiente

```
name = "Rufo"
age = 34
```

El lenguaje automáticamente detecta los tipos de datos de los valores y fija ese tipo a la variable. Si necesitamos convertir un valor de un tipo a otro hacemos uso de las palabras reservadas de Python.

```
age = int("20")
print(age) # Se imprime el entero 20

fraction = 0.1
intFraction = int(fraction) # Se obtiene 0
```

Claramente hay tipos de datos que no puede ser transformados en otro.

```
age = int("Valor")
```

1.1.1. Listas

Las listas son otro tipo de datos en Python, estas permiten agrupar un conjunto de valores o variables bajo un mismo nombre, por ejemplo.

```
dogs = ["Keiko", "Max"]
```

Podemos verificar si un elemento pertenece a una lista por medio del operador *in*, por ejemplo.

```
contain = ("Firulais" in dogs)  #Retorna False
```

Podemos crear una lista vacía.

```
myList = []
```

Podemos hacer referencia a elementos de las listas por medio índices, comenzando desde cero.

```
print(dogs[0]) #Imprime Keiko
print(dogs[1]) #Imprime Max
```

Para editar el valor de un elemento también hacemos uso de su indice.

```
dogs[1] = "Koby"
```

Podemos extraer una lista de otra, por ejemplo.

```
dogs = ["Keiko", "Max", "Koby", "Logan", "Canela"]

myDogs = dogs[0:2]
print(myDogs) #Imprime  ["Keiko", "Max"]

print(dogs[1:]) #Imprime ["Koby", "Logan", "Canela"]
```

Podemos obtener la cantidad de elementos de una lista por medio del método *len()*.

```
dogs = ["Keiko", "Max", "Koby", "Logan", "Canela"]
quantity = len(dogs)
print(quantity) #Imprime 5
```

Para agregar un elemento a una lista podemos usar el método *append()*, por ejemplo.

```
list = [2, 3, 5, 7]
list.append(11)
```

Si deseamos agregar una lista a otra lista podemos usar el método *extend()*.

```
list = [2, 3, 5, 7]
list.extend([11, 13, 17, 23])
```

O también usar el operado "+ =".

```
list = [2, 3, 5, 7]
list += [11, 13, 17, 23]
```

Para eliminar un elemento de la lista usamos el método *remove()*.

```
list = [2, 3, 5, 7]
list.remove(5)

print(list) #Imprime [2, 3, 7]
```

1.2. Operadores

Los operadores de Python son símbolos que utilizamos para ejecutar operaciones sobre valores y variables. Podemos dividir los operadores en función del tipo de operación que realizan.

- Operador de asignación
- Operadores aritméticos
- Operadores de comparación
- Operadores lógicos

El operador de asignación se utiliza para asignar un valor a una variable:

```
1   age = 8
```

O para asignar un valor de variable a otra variable

```
1   age = 8
2   anotherVariable = age
```

Python tiene varios operadores aritméticos como: + (Suma), − (Resta), ∗ (Multiplicacion), % (Resto), ∗∗ (Potencia), / (División) y // (División entera).

```
1   1 + 1 #2
2   2 - 1 #1
3   2 * 2 #4
4   4 / 2 #2
5   4 % 3 #1
6   4 ** 2 #16
7   4 // 2 #2
```

Los operadores numéricos generalmente retorna valores decimales (float), sin embargo, en los casos de resto (%) y división entera (//) se obtiene valores enteros si los argumentos son enteros, en caso contrario se obtiene float.

Algo interesante sobre el operador + es que permite "sumar" cadenas, ejemplo

```
1   string1 = "Py"
2   string2 = "thon"
3   print(string1 + string2) #Imprime Python
```

Python define algunos operadores de comparación:

- ==
- !=
- >

- <
- >=
- <=

Puedes usar esos operadores para obtener un valor booleano (**True** o **False**) según el resultado.

```
1   a = 1
2   b = 2
3   a == b #False
4   a != b #True
5   a > b #False
6   a <= b #True
```

Python nos proporciona los siguientes operadores booleanos

- not
- and
- or

Cuando se trabaja con atributos True o False, estos funcionan como AND, OR y NOT lógicos, y se utilizan a menudo en la evaluación de expresiones condicionales if.

```
1   condition1 = True
2   condition2 = False
3
4   not condition1 #se obtiene False
5   condition1 and condition2 #se obtiene False
6   condition1 or condition2 #se obtiene True
```

1.3. Números

Los números en Python pueden ser de tres tipos: entero (int), flotantes o reales (float) y complejos (complex).

Los números enteros se representan usando la clase int. La cual nos ayuda definir de manera literal de la forma

```
1   age = 8
```

Para los números flotantes definimos de manera literal de la forma

```
1   fraction = 0.1
```

Para los complejos usamos la forma

```
1   complexNumber = 2+3j
```

O usar el nombre reservado complex

```
complexNumber = complex(2, 3)
```

Podemos acceder a los valores reales e imaginarios por medio del operador de acceso[1] "." (punto).

```
complexNumber.real #2.0
complexNumber.imag #3.0
```

La biblioteca estándar de Python ofrece funciones y constantes de utilidad matemática que son más precisas que los operadores normales.

- El paquete *math* proporciona funciones y constantes matemáticas generales
- El paquete *cmath* proporciona utilidades para trabajar con números complejos.
- El paquete *decimal* proporciona utilidades para trabajar con decimales y números de punto flotante.
- El paquete *fractions* proporciona utilidades para trabajar con números racionales

Por ejemplo, en el paquete *math* ofrece funciones como "pow", "round", "abs", "factorial" entre otras muchas.

```
value = pow(2, 10) #2^(10) = 1024
sqrtOfInteger2 = pow(2, 0.5) #1.414213562...


value2 = round(sqrtOfInteger2) #1
value3 = round(sqrtOfInteger2, 3) #1.414
```

1.4. Sentencias de control

Lo interesante de hacer con los booleanos, y con las expresiones que devuelven un booleano en particular, es que podemos tomar decisiones y tomar diferentes caminos según su valor True o False.

A esto se le conoce como bifurcaciones ya que genera dos opciones en cómo puede ejecutar el programa, dando como resultado una toma de decisiones. En Python lo hacemos usando la declaración *if*:

```
condition = True

if (condition == True):
    # Hacer algo
```

[1] El operador de accesos toma relevancia en la orientación de objecto.

Cuando la condición se resuelve como True, como en el caso anterior, su bloque se ejecuta. Pero ¿qué es un bloque?, un bloque es la parte que tiene una identación (sangría) de un nivel[2] a la derecha, por ejemplo

```
1   condition = True
2
3   if (condition == True):
4       print("The condition")
5       print("was true")
```

El bloque puede estar formado por una sola línea, o por varias líneas, y finaliza cuando se vuelve al nivel de identación anterior:

```
1   condition = True
2
3   if (condition == True):
4       print("The condition")
5       print("was true")
6
7   print("Outside of the if")
```

En combinación con *if*, puede tener un bloque *else*, que se ejecuta si la prueba de condición de *if* da como resultado False.

```
1   condition = True
2
3   if (condition == True) :
4       print("The condition")
5       print("was True")
6   else:
7       print("The condition")
8       print("was False")
```

Y puedes tener diferentes casos vinculados con *elif*, que se ejecutan si la comprobación anterior fue falsa.

```
1   condition = True
2   name = "Roger"
3
4   if (condition == True):
5       print("The condition")
6       print("was True")
7   elif (name == "Roger"):
8       print("Hello Roger")
9   else:
10      print("The condition")
11      print("was False")
```

[2]De cuatro espacios generalmente.

El segundo bloque en este caso se ejecuta si la condición es False y el valor de la variable de nombre es "Roger". En una declaración **if**, puede tener solo una comprobación *if* y *else*, pero varias series de comprobaciones *elif*.

```
1   condition = True
2   name = "Roger"
3
4   if (condition == True):
5       print("The condition")
6       print("was True")
7   elif (name == "Roger"):
8       print("Hello Roger")
9   elif (name == "Syd"):
10      print("Hello Syd")
11  elif (name == "Flavio"):
12      print("Hello Flavio")
13  else:
14      print("The condition")
15      print("was False")
```

if y *else* también se pueden usar en formato en línea, lo que nos permite devolver un valor u otro en función de una condición.

```
1   a = 2
2   result = 2 if (a % 2 != 0) else 10*a
3   print(result) # Imprime 20
```

1.5. Cíclos

Los cíclos o bucles son una parte esencial de la programación. Ya que nos permite realizar operaciones sobre un conjunto de valores con número menor de código. En Python tenemos dos tipos de cíclos: *while* y *for*.

1.5.1. While

El cíclo *while* se definen utilizando la palabra clave while y repiten su bloque hasta que la condición se evalúa como False.

```
1   condition = True
2
3   while (condition == True):
4       print("The condition is True")
```

Este es un cíclos infinito, nunca termina porque la condición no cambia. Esto producirá un error por desbordamiento de memoria.

Por lo cual es necesario que el bloque del cíclo tenga un caso donde el la condición cambie. Veamos esto, deteniendo el cíclo justo después de la primera iteración:

```
condition = True

while (condition == True):
    print("The condition is True")
    condition = False

print("After the loop")
```

En este caso, se ejecuta la primera iteración, ya que la prueba de condición se evalúa como True, y en la segunda iteración la prueba de condición se evalúa como False, por lo que el control pasa a la siguiente instrucción, después del cíclo. Es común tener un contador para detener la iteración después de una cierta cantidad de ciclos:

```
count = 0

while (count < 10):
    print("The condition is True")
    count = count + 1

print("After the loop")
```

1.5.2. For

Mediante un cíclo *for* podemos indicarle a Python que ejecute un bloque una cantidad de veces predeterminada, por adelantado y sin necesidad de una variable y una condición independientes para comprobar su valor. Por ejemplo, podemos iterar los elementos de una lista como una cadena.

```
cadena = "Esta es una cadena."

for item in cadena:
    print(item) #Se imprime cada caracter en la cadena
```

1.5.3. Break y Continue

Tanto los cíclos while como los for pueden interrumpirse dentro del bloque, utilizando dos palabras clave especiales: *break* y *continue*.

- La sentencia *continue* detiene la iteración actual y le indica a Python que ejecute la siguiente.
- La sentencia *break* detiene el cíclo por completo y continúa con la siguiente instrucción después del final del cíclo.

Por ejemplo, en el siguiente código no se imprime ninguna letra “a”.

```
1   cadena = "Esta es una cadena."
2
3   for item in cadena:
4       if (item == "a"):
5           continue
6
7       print(item)
```

En el siguiente código se imprime sin llegar a la letra “c”.

```
1   cadena = "Esta es una cadena."
2
3   for item in cadena:
4       if (item == "c"):
5           break
6
7       print(item)
```

1.6. Funciones

Una función nos permite crear un conjunto de instrucciones que podemos ejecutar cuando sea necesario. Las funciones son esenciales en Python y en muchos otros lenguajes de programación para crear programas significativos, ya que nos permiten descomponer un programa en partes manejables, promueven la legibilidad y la reutilización del código. Aquí hay un ejemplo de función llamada hello que imprime “Hola”:

```
1   def hello():
2       print('Hola')
```

Esta es la definición de la función. Hay un nombre (hello) y un cuerpo, el conjunto de instrucciones, que es la parte que sigue a los dos puntos y está con una identación de un nivel a la derecha. Para ejecutar esta función, debemos llamarla. Esta es la sintaxis para llamar a la función:

```
1   hello()
```

Podemos ejecutar esta función una vez o varias veces, el nombre de la función, hello, es muy importante. Debe ser descriptivo, para que cualquiera que la llame pueda imaginar lo que hace la función.

Una función puede aceptar uno o más parámetros:

```
1   def hello(name):
2       print('Hello ' + name + '!')
```

En este caso llamamos a la función pasando el argumento

```
1   hello("Roger")
```

Así es como podemos aceptar múltiples parámetros:

```
1   def hello(name, age):
2       print('Hello ' + name + ', you are ' + str(age)
3                                               + ' years old!')
```

En este caso llamamos a la función pasando un conjunto de argumentos:

```
1   hello("Roger", 20)
```

Una función puede devolver un valor mediante la sentencia *return*. Por ejemplo, en este caso, devolvemos el nombre del parámetro:

```
1   def hello(name):
2       print('Hello ' + name + '!')
3       return name
```

Cuando la función cumple con la declaración de retorno, la función finaliza.

Puede devolver varios valores utilizando valores separados por comas:

```
1   def hello(name):
2       print('Hello ' + name + '!')
3       return name, 'Roger', 8
```

En este caso, al llamar a hello('Syd'), el valor de retorno (nota: no lo que está impreso en la pantalla, sino el valor de retorno) es una tupla que contiene esos 3 valores: ('Syd', 'Roger', 8):

```
1   def hello(name):
2       print('Hello ' + name + '!')
3       return name, 'Roger', 8
4
5   print(hello('Syd')) #('Syd', 'Roger', 8)
```

1.7. Ejercicios

Ejercicio 1.1. Análisis de los valores float y enteros, probar los siguientes valores en un archivo .py con la ayuda del método print.

- Calcular 12 ** 34 y 12.0 ** 34.0 (notar como Python representa los números por medio de notación científica)
- Calcular 1 / 12 ** 34 y ver como se representa.
- Asignar los números $12{,}3 \times 10^{45}$ y $12{,}3 \times 10^{-45}$ a un variable usando la notación científica de Python (con e)[3].
- Calcular 123 ** 456.

[3]No confundir con la constante de Euler, la e indica exponente base 10.

- Calcular 123.0 ** 456.0.
- Calcular 123.0 ** 456 y 123 ** 456.0.

Ejercicio 1.2. Usando ==, <= y >= comparar

- 123 con 123.0
- 123 con 123 + 1.0e-10
- 123 con 123 + 1.0e-20

Ejercicio 1.3. Ver la diferencias entre

- print(123456) y print(123, 456)
- print("Un comentario") y print("Un", "comentario")

Ejercicio 1.4. Hacer un programa que calcule la media aritmética de 6 números que se ingresan por pantalla.

Ejercicio 1.5. Hacer un programa que pida dos ángulos de un triángulo por pantalla e imprima si el triángulo es acutángulo, rectángulo y obtuso.

Ejercicio 1.6. Hacer un programa que pida el radio de una circunferencia y calcule e imprima su área y perímetro.

Ejercicio 1.7. Dado un triángulo rectángulo ABC con $AB = BC$, hacer un programa que imprima el valor del ángulo $\angle ABC$ si se ingresa las medidas de AC y la altura desde B hacia AC.

Ejercicio 1.8. Hacer un programa que pida los lados de un triángulo y calcule el inradio.

Ejercicio 1.9. Hacer un programa que pida dos números enteros y calcule su MCD, por medio del algoritmo de Euclides.

Ejercicio 1.10. Hacer un programa que pida por pantalla un número entero n mayor a cero y menor a 100. Que pida n números por pantalla y calcule la media geometríca.

2. Programación orientada a objetos

– Pendiente –