Universidad Nacional de Rosario (UNR)
Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Curso introductorio a la Programación

Aprendiendo a programar desde cero utilizando Python

(Módulo I)

Autor: Lic. Leonardo Bartocci

Desarrollado con la expectativa de ayudar a dar el primer paso a toda persona que desee introducirse en el mágico mundo de la programación

Septiembre del 2019

Índice general

Capítulo 1

Introducción

Este curso está dirigido a quienes desean dar el primer paso en el mundo de la programación.

El lenguaje utilizado como herramienta para el aprendizaje será *Python*. Pero se intentará utilizar sentencias de de usos comunes a la mayoría de los lenguajes de uso masivo de forma que quién realice este curso luego pueda sentirse familiarizado con las sentencias usuales de cualquier otro lenguaje que necesite utilizar en el futuro.

Es importante que el estudiante entienda desde el primer momento que este curso tiene como objetivo brindarle una introducción informal a la programación en general y que *Python* es sólo el lenguaje elegido como herramienta, pero la herramienta podía haber sido cualquier otro lenguaje de programación. Resumiendo, este curso intenta enseñar a programar independientemente del lenguaje de programación a utilizar.

En este documento (correspondiente al Módulo I) se presentan los conceptos y nociones básicas de programación.

Se pretende que luego de leer este documento el estudiante haya adquirido la motivación y los conocimientos necesarios para continuar con el **módulo II** en donde realizará una serie de ejercicios reales de programación.

1.1. Pre-requisitos

Como pre-requisitos se asume que el estudiante tiene algunos conocimientos generales sobre el sistema operativo que utiliza como son el manejo de archivos y la

instalación y gestión de aplicaciones de uso general.

Se requiere además tener instalado el entorno de *Python*. En caso de no tenerlo aún, se recomienda acceder al siguiente link para ver las instrucciones paso a paso para la descarga y la instalación: Instrucciones para descargar e instalar el entorno Python

1.2. Consideraciones generales

Los temas se irán presentando de forma gradual y sin entrar en detalles formales rigurosos, mas bien se pretende dar una introducción informal y varios ejemplos y ejercicios para que el lector valla entendiendo la utilidad y asimilando la forma de aplicación de cada tema.

1.3. Nomenclatura para nombres de archivo de programas

Respecto de los ejercicios, algunos se indican realizar directamente sobre el IDLE y otros proponen escribir un programa para guardar y ejecutar. Un programa es un archivo cuyas lineas son instrucciones de programación escritas en un lenguaje determinado, en nuestro caso Python.

En el enunciado de cada ejercicio donde se solicita escribir un programa, se indica la palabra **Ejercicio_X_Y**. El programa correspondiente a ese ejercicio se almacena en el archivo de nombre **ejer_X_Y.py** dentro de la carpeta Scripts del directorio de instalación de Python.

Por ejemplo, si se presenta el ejercicio **Ejercicio_4_1** entonces el programa se debe almacenar con el nombre **ejer_4_1.py**.

Capítulo 2

Conceptos básicos

En este capítulo se introduce un conjunto de conceptos básicos que se utilizan en la jerga de la programación. No es necesario que el lector comprenda todos los conceptos en una primera lectura ya que se espera que los valla asimilando en la medida que progrese con el curso.

2.1. Sentencias

Una **sentencia** es una instrucción de programación indivisible, esto generalmente es una línea de código. Por ejemplo la siguiente línea de código es una sentencia *Python* que asigna un valor a una variable.

```
varNombre="un valor"
```

2.2. Programas

Un **programa** es un conjunto de sentencias escritas en un cierto orden para realizar una tarea determinada. En Python un programa se guarda en un archivo de extensión **py**. Al contenido de un archivo **py** se lo llama código fuente.

2.3. Variables

Los programas principalmente manipulan datos, los datos se almacenan en elementos del programa llamados **variables**. Las variables se identifican por un nombre y contienen un valor determinado que se le es asignado durante la ejecución del programa. Por ejemplo, si durante la ejecución del programa se ejecuta la siguiente sentencia:

```
edad=28
```

Entonces la variable *edad* pasará a tener el valor *28*.

2.4. Funciones

Las **funciones** contienen un conjunto de instrucciones de uso común y se puede reutilizar las veces que sea necesario dentro de los programas. Phyton provee de forma pre-definida un gran número de funciones y a su vez el usuario puede definir otras tantas funciones como desee.

Una función puede o no recibir valores de entrada y realizar alguna acción en función de esos valores. Los valores de entrada se llaman **argumentos** o **parámetros** de la función. Una función se invoca por su nombre y una lista de uno o más parámetros delimitados por paréntesis y separados por coma.

Por ejemplo, la siguiente sentencia invoca a la función *calcularArea* pasando como parámetros los números 3 y 2.

```
calcularArea(3, 2)
```

2.5. Código compilado vs código interpretado

En muchos lenguajes de programación, como por ejemplo Java, el código fuente no se ejecuta directamente sino que debe ser previamente compilado y lo que se ejecuta es el *código compilado*. El proceso de compilación lo que hace es interpretar el código escrito por el programador y generar un *código a nivel de máquina*. El código compilado es más eficiente para la máquina pero ilegible para las personas.

Cuando el código no es compilado se dice que se ejecuta en modo interpretado. A los programas que ejecutan en modo intérprete también se los suele llamar *Scripts*.

A los programas Python se los puede ejecutar tanto en modo interpretado como en modo compilado, es por ello que en la web es usual que se mencione Script Python para hacer referencia a un programa Python.

Por simplicidad en este curso comenzaremos utilizando el modo interpretado.

Capítulo 3

Tipos de datos primitivos

En un programa principalmente se manipulan datos, los datos se almacenan en **variables**. Todos los lenguajes masivos de programación poseen al menos los tipos de datos primitivos **Numérico, String y Booleano**. Luego cada lenguaje puede presentar otros tipos de datos adicionales. En las siguientes secciones se enumeran los tipos de datos primitivos de *Python* más comunes.

3.1. Tipos de datos numéricos

Los tipos de datos numéricos que utilizaremos son **enteros (int) y flotantes (float)**. El tipo entero representa un número entero y el tipo flotante representa un número con punto decimal.

Por ejemplo, podemos escribir la siguiente instrucción para definir una variable de nombre *varEntera* que almacena el número 10:

```
varEntera=10
```

Y de la misma forma podemos definir la variable de nombre *varFlotante* que almacene el número 10,5 (observar que se debe utilizar el punto como separador decimal)

```
varFlotante=10.5
```

Ejercicio 3_1: Ingresar lo siguiente desde el IDLE

```
>>> varEntera=10
>>> varFlotante=10.5
>>> varEntera
>>> varFlotante
```

Observar que al tipear *varEntera* el intérprete retorna el valor que contiene la variable varEntera y lo mismo sucede con la variable varFlotante.

La figura 3.1 ilustra el ejercicio realizado desde el IDLE.

Figura 3.1: Ejer_3_1

3.2. Tipo de dato String

Un string es una cadena de caracteres. Los strings se declaran entre comillas dobles de la siguiente manera:

```
varString="Este es un string"
```

Ejercicio 3_2: Definir la variable varString asignándole el valor "Este es un string" en el IDLE y luego introducir el nombre de la variable para inspeccionar su contenido.

Figura 3.2: Ejer_3_2

3.3. Tipo de dato Booleano

Una variable de tipo booleano puede almacenar los valores **True** o **False** (**verdadero** o **falso**). Este tipo de variables se utilizan principalmente en sentencias condicionales

que veremos más adelante. De momento solo nos interesa aprender a declararlas. A modo de ejemplo se indica como declarar las variables *varBooleanT* y *varBooleanF* asignándoles valores *True* y *False* respectivamente.

```
varBooleanT=True
varBooleanF=False
```

Ejercicio_3_3: utilizando el IDLE definir las variables *varBoolean1* y *varBoolean2* asignándoles valores *True* y *False* respectivamente y luego explorar el valor de las mismas.

```
>>> varBoolean1=True
>>> varBoolean2=False
>>> varBoolean1
True
>>> varBoolean2
False
```

3.4. Tipo de datos Lista

Además de los tipos de datos simples enumerados hasta aquí, Python presenta algunos tipos de datos complejos como son las listas. Una lista es una colección de elementos (datos) de cualquier tipo (numérico, string, booleano, etc), incluso un elemento de una lista puede ser otra lista. Una lista se define utilizando [ ] como delimitadores y los elementos se separan por comas. A continuación la definición de una lista.

```
listaEjemplo=["String", 10, True]
```

Esta sentencia define la lista de nombre listaEjemplo que tiene los valores "String" de tipo string, 10 de tipo numérico y True de tipo booleano.

Por ejemplo, se podría definir una lista indicando el nombre, la edad, el sexo, el peso y la altura de una persona de la siguiente manera:

```
listaPersona=["Juan", 22, "M", 72.5, 1.65]
```

Se accede a un dato de una lista a través de su índice, esto es su posición. El primer elemento corresponde al índice 0. Por ejemplo para conocer el tercer elemento de la lista anterior se puede ejecutar lo siguiente:

```
>>> listaPersona=[2]
'M'
```

Las listas son **mutables**, esto significa que sus elementos se pueden modificar en cualquier momento. Por ejemplo, podemos modificar el primer elemento de la lista anterior de la siguiente manera:

```
>>> listaPersona[0]="Jose"
>>> listaPersona
["Jose", 22, "M", 72.5, 1.65]
```

3.4.1. Listas anidadas

Una lista puede contener a otra lista como elemento. El siguiente ejemplo define en primer lugar las listas *persona1* y *persona2* con dos elementos que indican el nombre y la edad de una persona. Y luego define la lista *personas* que contiene como elementos a las listas *persona1* y *persona2*.

```
>>> persona1=['Juan', 23]
>>> persona2=['Pedro', 21]
>>> personas=[persona1, persona2]
>>> personas
[['Juan', 23], ['Pedro', 21]]
```

Para acceder a un elemento de una lista anidada se requiere indicar en primer lugar el índice del elemento de la lista principal y en segundo lugar el índice del elemento de la sub-lista. Por ejemplo, si se quiere acceder al primer elemento de *personas2* se puede utilizar la siguiente instrucción:

```
>>> personas[1][0]
'Pedro'
```

Ejercicio_3_4_1: utilizando el IDLE definir listas anidadas y explorar los distintos elementos de esas listas.

3.5. Otros tipos de datos de Python

Además de los tipos de datos mencionados, Python cuenta con otros tipos de datos nativos como son las tuplas, diccionarios y conjuntos. Por ser estos menos comunes que los restantes no se van a considerar en este módulo ya que como se remarcó anteriormente el propósito de este curso es que el lector aprenda a programar con sentencias y tipos de datos comunes a la mayoría de los lenguajes de programación.

Capítulo 4

Sentencias de iteración con el usuario

La miseración del usuario con el programa se realiza a través de entrada y salida de datos. En este capítulo se describen las funciones que se utilizan para solicitar datos de entrada al usuario y las que se utilizan para mostrar los datos de salida al usuario.

4.1. Función print (para la salida de datos)

En *Python* se utiliza la función **print** para mostrar información al usuario, esto es, imprimir texto en pantalla. Esta función puede recibir como parámetros una lista de valores e imprime un texto resultante de la concatenación de esos valores.

A continuación varios ejemplos de uso de la función **print**.

Ejemplo: imprime el string constante "Hola".

```
>>>print("Hola")
Hola
```

Ejemplo: imprime la concatenación de dos strings.

```
>>> print("Hola", "Juan")
Hola Juan
```

Ejemplo: imprime la concatenación del string constante "Holaz el valor de la variable nombre.

```
>>> nombre="Juan"
>>> print("Hola", nombre)
Hola Juan
```

Ejemplo: imprime la concatenación de un string constante, el valor de la variable nombre y otro string constante.

```
>>> nombre="Juan"
>>> print("Hola", nombre, ". ¿Cómo estás?")
Hola Juan. ¿Cómo estás?
```

Ejemplo: imprime la concatenación de un string constante y el valor del resultado de una expresión aritmética.

```
>>> print("1 + 2 es ", 1 + 2)
1 + 2 es 3
```

Ejercicio 4_1: Primer programa Phyton!!

Escribir un programa Phyton con el nombre de archivo *ejer_4_1.py* que defina las variables *nombre* con el valor *"Juan"*, *edad* con el valor *20* y muestre en pantalla "[nombre] tiene [edad] años de edad".

Por ser el primer programa se enumeran los pasos para editar el archivo *py* y ejecutarlo.

- Desde el IDLE: *File -> New File*
- Escribir las siguientes instrucciones en el nuevo archivo:
  - nombre="Juan"
  - edad=20
  - print(nombre, "tiene", edad, ".ªños de edad")
- Guardar el archivo: *File -> Save (en la carpeta Scripts con nombre ejer_4_1.py)*
- Ejecutar el programa: pulsar la techa *<F5>*

Al ejecutar el programa desde el IDLE se visualizará el siguiente resultado:

```
Juan tiene 20 años de edad
```

4.2. Función input (para la entrada de datos)

Comúnmente la función input se utiliza en una sentencia que pide un valor y lo almacene en una variable.

A modo de ejemplo, la siguiente sentencia pide la edad de una persona y almacena el dato ingresado en la variable *edad*.

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

Ejercicio 4_2: Escribir un programa con nombre de archivo ejer_4_2.py que pida el nombre y la edad de una persona y muestre el mensaje: nombre tiene edad años de edad.

```
nombre=input("Ingrese su nombre: ")
edad=input("Ingrese su edad: ")
print(nombre, "tiene", edad, "años")
```

Ejecutar el programa, ingresar nombre y edad y visualizar el resultado.

Capítulo 5

Comentarios

Los comentarios son bloques de texto que utilizados de forma conveniente pueden ser de gran ayuda al programador para entender y mantener el código de los programas. Generalmente un comentario indica qué es lo que hace una instrucción o un bloque de código. El intérprete de Python ignora los comentarios, o sea, que los comentarios no tienen ningún efecto en el momento de la ejecución del programa.

Los comentarios pueden ser escritos en una línea o en varias líneas, a estos últimos se los llama comentarios multi líneas.

5.1. Comentarios en una línea

Todo texto que aparezca luego del carácter "#ƨ hasta finalizar la línea de texto se considera un comentario. A continuación algunos ejemplos.

```
# Este es un comentario en una línea.

print("Hola")   # Este comentario aparece luego de una instrucción
```

5.2. Comentarios multi líneas

Los comentarios multi líneas se delimitan con tres comillas dobles. A continuación un ejemplo.

```
"""Este es un comentario
escrito en más de una línea"""
```

Ejercicio 5_1: Escribir un programa con el siguiente código y ejecutarlo varias veces cambiando los valores de entrada.

```
""" Mi primer programa con comentarios
Variables de entrada del programa (ingresadas por el usuario):
  - nombre: nombre de la persona
  - edad: años de edad de la persona

Salida del programa:
  - Imprime el texto "{nombre} tiene {edad} años de edad"
"""

# Pide los datos de entrada
nombre=input("Ingrese el nombre: ")
edad=input("Ingrese la edad: ")

# Salida
print(nombre, " tiene ", edad, " años de edad")
```

Capítulo 6

Operaciones sobre tipos de datos

En este capítulo se mencionan las operaciones elementales que se pueden realizar sobre los tipos de datos. Se explicará cómo inspeccionar el tipo de datos de una variable, cómo convertir de un tipo de dato a otro y los operadores y métodos que se pueden aplicar sobre cada tipo de datos.

6.1. Chequeo del tipo de dato de una variable

Más adelante se verá la importancia de conocer el tipo de datos de una variable. En otros lenguajes de programación como Java, el tipo de datos de una variable se indica explícitamente al declarar la variable. En cambio en *Python* el tipo de dato es inferido por el intérprete de acuerdo al valor que se le asigne a la variable.

Por ejemplo, si se asigna un valor entero a una variable, *Python* asume que la variable es de tipo entero. Si se le asigna un string, *Python* asume que la variable es de tipo string y así con todos los tipos.

Para chequear el tipo de datos de una variable se utiliza la función *type*. Dicha función recibe el nombre de la variable como parámetro e indica de qué tipo (o clase) es esa variable.

```
>>>varEntero=10
>>>type(varEntero)
<class 'int'>
```

Ejercicio 6_1: Desde el IDLE asignar un valor de cada tipo (entero, flotante, string, booleano, lista) a una variable y chequear el tipo de la variable con la función *type*.

```
>>> varTest=10
>>> type(varTest)
<class 'int'>

>>> varTest=10.5
>>> type(varTest)
<class 'float'>

>>> varTest="hola"
>>> type(varTest)
<class 'str'>

>>> varTest=bool
>>> type(varTest)
<class 'type'>

>>> varTest=(10, "hola")
>>> type(varTest)
<class 'tuple'>
```

6.2. Conversiones de datos

Muchas veces puede ser necesario convertir de un tipo de datos a otro. Por ejemplo, pasar una variable de tipo str a int para poder aplicar sobre esta un operador aritmético.

Python brinda un conjunto de funciones de conversión de datos que se enumeran en la siguiente tabla.

Tabla 6.1: Tabla de funciones de conversión de tipo de datos.

Función	**Convierte a**
`str()`	str – cadena de caracteres
`int()`	int - entero
`float()`	float - flotante
`bool()`	bool - booleano
`list()`	list - lista

Al aplicar cualquiera de estas funciones sobre una variable, la función retornará el valor convertido al tipo indicado. Luego ese valor puede ser utilizado en una expresión o reasignarse a la misma variable haciendo que esta cambie de tipo. A continuación algunos ejemplos.

Ejemplo: se define la variable *varNumero* asignándole un valor de tipo string y luego convierte el tipo de la variable de *string* a *int* para poder utilizarlo en la suma.

```
>>> varNumero="10"
>>> type(varNumero)
<class 'str'>
>>> varNumero=int(varNumero)
>>> type(varNumero)
<class 'int'>
>>> varNumeroMas5 = varNumero + 5
>>> print(varNumeroMas5)
15
```

6.2.1. Errores de conversiones de tipo de datos

Cuando se aplica una conversión de tipo de datos sobre un valor es necesario tener en cuenta que el valor sea compatible con la función de conversión que se aplique. Por ejemplo, para convertir de un tipo de dato *string* a uno *numérico* es necesario que el valor a convertir represente un número.

Para aclarar la idea, el siguiente ejemplo muestra el error que arroja el intérprete al intentar convertir a entero un string que tiene un valor que no representa un número entero.

```
>>> varTest="un texto no entero"
>>> int(varTest)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    int(varTest)
ValueError: invalid literal for int() with base 10: 'un texto no entero'
```

El mensaje indica que el valor "un texto no entero" es inválido para aplicarle la función de conversión *int()*.

De la misma forma que al aplicar una función de conversión de datos, el intérprete valida el tipo de datos al intentar aplicar cualquier otra operación. El siguiente

ejemplo muestra el error que se obtiene al intentar aplicar la operación suma sobre un valor de tipo entero y otro de tipo string.

```
>>> varTest="10"
>>> 5 + varTest
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    5 + varTest
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

El mensaje indica que el operador "+" no puede ser aplicado sobre los tipos entero y string. Observar que el valor de la variable *varTest* es de tipo string. Luego para poder aplicar la suma es necesario convertir el valor a int con la función de conversión *int(varTest)*.

6.3. Operadores y funciones sobre tipos de datos

En esta sección se introducen las operaciones posibles de realizar sobre los tipos de datos. Estas operaciones se realizan a través del uso de *operadores* y *funciones* que provee el lenguaje. Dado que son provistos por el lenguaje se los denominan *operadores y funciones nativas*. El programador puede definir sus propias *funciones* a través de estos elementos nativos pero eso se explicará más adelante.

6.3.1. Operadores

Se llaman *operadores* a los símbolos (signos o palabras reservadas) que el intérprete de Python identifica como una operación a realizar sobre datos de un tipo determinado. Por ejemplo el símbolo "+" aplicado a dos valores numéricos realiza la suma algebraica entre esos dos números. En secciones siguientes se irán enumerando los operadores nativos provistos por Python para cada tipo de dato.

6.3.2. Métodos

Además de operadores, el lenguaje Python brinda *métodos nativos*. Estos métodos son una especie de funciones aplicables sobre determinados tipo de datos. A continuación un ejemplo de un método aplicable a un tipo de dato string.

```
>>>varStr="abc def ghi jkl"
>>> varStr.upper()
'ABC DEF GHI JKL'
```

En este ejemplo se aplica el método *upper()* al valor de la variable varStr. El método upper() convierte todos los caracteres del string a mayúsculas (upper case).

6.3.3. Noción de polimorfismo de operadores

Un mismo operador puede tener connotaciones diferentes de acuerdo al tipo de datos que sobre el que se aplique, por ejemplo el operador "+" aplicado a tipos numéricos realiza la suma algebraica pero el mismo símbolo aplicado a tipos string realiza la concatenación de dos strings. La concatenación es la unión de dos strings en un string mayor. A los operadores y/o funciones aplicables a distintos tipos de datos se los denomina *operadores polimórficos*.

En las secciones siguientes se presentarán los operadores y métodos más comunes brindados por Python.

6.4. Operadores sobre tipos de dato numérico

La siguiente tabla enumera los operadores disponibles para operar sobre tipos de datos numéricos. Estos son operadores aritméticos.

Tabla 6.2: Tabla de operadores sobre datos numéricos.

Operador	**Operación**
+	suma
-	resta
-	negativo
*	multiplicación
**	exponente
/	división
/	división entera
%	residuo

Ejercicio 6_2: Desde el IDLE asignar el valor 3 a la variable *num1* y el valor 2 a la variable *num2*. Luego hacer un print para mostrar el resultado de aplicar cada uno de los operadores a esas dos variables.

```
>>> n1=3
>>> n2=2
>>> print("n1+n2: ", n1+n2)
n1+n2:  5
>>> print("n1-n2: ", n1-n2)
n1-n2:  1
>>> print("-n1: ", -n1)
-n1:  -3
>>> print("n1*n2: ", n1*n2)
n1*n2:  6
>>> print("n1**n2: ", n1**n2)
n1**n2:  9
>>> print("n1/n2: ", n1/n2)
n1/n2:  1.5
>>> print("n1//n2: ", n1//n2)
n1//n2:  1
>>> print("n1%n2: ", n1%n2)
n1%n2:  1
```

6.5. Operadores y métodos sobre tipo de dato string

En esta sección se enumeran los operadores y funciones comunes sobre tipos de datos string.

6.5.1. Operador de concatenación (+)

El símbolo "+" aplicado a dos valores string realiza la *concatenación* de los dos valores.

El siguiente ejemplo muestra el resultado de aplicar la *concatenación* sobre dos variables de tipo string.

```
>>> varStr1="abc"
>>> varStr2="def"
>>> varConcat=varStr1+varStr2
>>> print(varConcat)
abcdef
```

A continuación se presenta otro ejemplo donde se concatenan la variable *varNombre*, el string constante "tiene ", la variable *varEdad* y el string constante "de edad". El resultado de la concatenación se asigna a la variable *varMensaje* y luego se imprime el mensaje.

```
>>> varNombre="Juan"
>>> varEdad="35"
>>> varMensaje=varNombre + " tiene " + varEdad + " de edad"
>>> print(varMensaje)
Juan tiene 35 de edad
```

Y como último ejemplo sobre el operador "+" se muestra el error que produce cuando se lo aplica sobre valores de distintos tipos.

```
>>> varStr1='abc'
>>> varStr2='def'
>>> varNum1=1
>>> varNum2=2
>>> # Aplicados sobre números realiza la suma
>>> varNum1 + varNum2
3
>>> # Aplicados sobre string realiza la concatenación
>>> varStr1 + varStr2
'abcdef'
>>> # Aplicado sobre un tipo string y un tipo numérico produce error de tipo
>>> varStr1 + varNum1
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    varStr1 + varNum1
TypeError: can only concatenate str (not "int") to str
>>> # Para resolver el error se puede convertir el número a strig con la función de conversión str()
>>> varStr1 + str(varNum1)
'abc1'
```

6.5.2. Métodos sobre string

La siguiente tabla enumera algunos de los métodos más comunes sobre strings. Para conocer la lista completa referirse a la documentación oficial de Python.

Tabla 6.3: Tabla de métodos sobre tipo de dato string.

Método	**Función**
`capitalize()`	Transforma a mayúscula el primer caracter del string
`count()`	Retorna el número de veces que aparece un valor (substring) en el string
`find()`	Busca un substring dentro del string y retorna la posición donde este aparece
`lower()`	Transforma a minúscula todos los caracteres del string
`lstrip()`	Retorna el string sin los espacios en blanco de la izquierda
`replace()`	Reemplaza un substring por otro dentro del string
`rstrip()`	Retorna el string sin los espacios en blanco de la derecha
`strip()`	Retorna el string sin los espacios en blanco de la izquierda ni de la derecha
`upper()`	Transforma a minúscula todos los caracteres del string

En los siguientes ejemplos se muestra el resultado de aplicar cada uno de los métodos enumerados.

```
>>> # MétodoS capitalize(), lower(), upper()
>>> varNombre="juan"
>>> varNombre.capitalize()
'Juan'
>>> varNombre.lower()
'juan'
>>> varNombre.upper()
'JUAN'
```

```
>>> # Método count()
>>> varStr="a ab abc abcd"
>>> varStr.count("ab")
2

>>> # Método find()
>>> varStr.find("abc")
5  # el string 'abc' aparece en la posición 5, tener en cuenta que la primer posición es 0.
```

```
>>> # Métodos lstrip(), rstrip(), strip()
>>> varStr="    abc def   "
>>> varStr.lstrip()
'abc def   '
>>> varStr.rstrip()
'    abc def'
>>> varStr.strip()
'abc def'
```

```
>>> # Método replace
>>> varStr='a ab abc'
>>> varStr.replace('ab', '12')
'a 12 12c'
```

6.6. Métodos sobre tipo de dato Lista

Recordar que una lista es una colección de elementos (datos) de cualquier tipo (numéerico, string, booleano, etc), incluso un elemento de una lista puede ser otra lista. La siguiente tabla enumera los métodos más comunes sobre listas.

Tabla 6.4: Tabla de métodos sobre tipo de dato Lista.

Método	**Función**
`append()`	Adiciona un elemento al final de la lista
`clear()`	Remueve todos los elementos de la lista
`copy()`	Retorna una copia de la lista
`count()`	Retorna la cantidad de veces que se repite un elemento en la lista
`index()`	Busca un elemento dentro de la lista y retorna la posición de ese elemento
`insert()`	Inserta un elemento en la posición indicada
`pop()`	Remueve un elemento de la posición indicada
`remove()`	Remueve los items de la lista con cierto valor indicado
`reverse()`	Reversa el orden de la lista
`sort()`	Ordena los elementos de la lista

A continuación varios ejemplos ilustrativos de la aplicación de estos métodos.

```
>>> varLista=['a', 'b', 'c', 'a', 'ab', 'abc']
>>> varLista
['a', 'b', 'c', 'a', 'ab', 'abc']
>>> # Método append()
>>> varLista.append('abcd')  # Adiciona el elemento 'abcd' al final de la lista
>>> varLista
['a', 'b', 'c', 'a', 'ab', 'abc', 'abcd']
>>> # Método remove()
>>> varLista.remove('abcd')   # Remueve el elemento 'abcd'
>>> varLista
['a', 'b', 'c', 'a', 'ab', 'abc']
>>> # Método pop()
>>> varLista.pop(5)   # Retorna el elemento de la posición 5 y lo remueve de la lista
'abc'
>>> varLista
['a', 'b', 'c', 'a', 'ab']
>>> # Método insert()
>>> varLista.insert(4, 'aa') # Inserta elemento 'ab' en la posición 5
>>> varLista
['a', 'b', 'c', 'a', 'aa', 'ab']
>>> # Método clear()
>>> varLista.clear() # Remueve todos los elementos de la lista
>>> varLista
[]
```

```
>>> varLista=['b', 'a', 'c', 'b']
>>> # Método copy()
>>> copyLista=varLista.copy()  # guarda en copyLista una copia de varLista
>>> copyLista
['b', 'a', 'c', 'b']
>>> # Método count()
>>> varLista.count('b')  # cuenta las ocurrencias del elemento 'b'
2
>>> # Método index()
>>> varLista.index('c')  # retorna el indice donde aparece el valor 'b'
2
```

```
>>>varLista=[3,2,4,1]

>>> # Método reverse()
>>> varLista.reverse()  # reversa el orden de los elementos dentro de la lista
>>> varLista
[1, 4, 2, 3]
>>> # Método sort()
>>> varLista.sort()  # ordena los elementos dentro de la lista
>>> varLista
[1, 2, 3, 4]
```

6.7. Operadores sobre tipo de dato booleano

Recordar que existen sólo dos valores booleanos que son *True* y *False*. Los operadores booleanos operan sobre estos valores y retornan también un valor booleano *True* o *False*. A estos operadores se los denomina también *operadores lógicos* porque realizan comparaciones lógicas (and, or, not). En un principio puede que el estudiante no vea demasiada utilidad en estos operadores pero en el próximo capítulo comprenderá que son esenciales para la programación.

Tabla 6.5: Tabla de operadores sobre tipos booleanos.

Operador	**Función**
`and`	Retorna True si ambos valores comparados son True
`or`	Retorna False si uno de los valores comparados es True
`not`	Aplica sobre un único valor y retorna True si el valor es False

A continuación algunos ejemplos de uso de estos operadores.

```
>>> varVerdadero=True
>>> varFalso=False
>>> # Operador and
>>> varVerdadero and varFalso
False
>>> # Operador or
>>> varVerdadero or varFalso
True
>>> # Operador not
>>> not varVerdadero
False
>>> not varFalso
True
```

6.8. Operadores de comparación

Los operadores de comparación se utilizan para comparar valores. Por ejemplo para comparar si el valor de la variable numérica *var1* es menor al valor de la variable numérica *var2*. Al ser aplicado, un operador de comparación retorna el valor *True*

(verdadero) si se cumple la condición de comparación o retorna *False* (falso) en caso contrario.

La aplicación más común de estos operadores es sobre tipos de datos numéricos pero pueden ser utilizados para cualquier tipo de datos. El más utilizado para otros tipos de datos es seguramente el comparador de igualdad.

Tabla 6.6: Tabla de operadores de comparación.

Operador	**Operación**
`==`	Igual - Retorna True si los valores comparados son iguales.
`!=`	Diferentes - Retorna True si los valores comparados son diferentes.
`>`	Mayor a - Retorna True si el valor de la izquierda es mayor al de la derecha
`<`	Menor a - Retorna True si el valor de la la izquierda es menor al de la derecha
`>=`	Mayor o igual a
`<=`	Menor o igual a