Sentencias básicas y datos simples

Unidad 2

Apunte de cátedra
2do Cuatrimestre 2023

Pensamiento computacional (90)
Cátedra: Camejo

.UBAXXI

1. Sentencias básicas

Habiendo revisado ya algunos conceptos sobre programación, ahora podemos centrarnos en la herramienta que vamos a emplear: **Python**, sobre cómo hacer un programa sencillo, cómo interactuar con el usuario, y más. ¿Estamos listos?

1.1. Flujo de Control de un Programa

Para desarrollar un programa en Python 3.x escribiremos una colección de sentencias en orden de ejecución, de arriba hacia abajo; una en cada renglón, respetando la sintaxis de cada una. La secuencia en que las sentencias se ejecutan se llama **Flujo de Control de un Programa** (**FCP**, para nosotros , de ahora en más).

La ejecución de la **primera** a la **última**, una por una, es un **Flujo de Control Estándar** en Python. Es decir, que si el siguiente bloque fuese un programa de Python:

```
Esto se ejecuta primero
Esto se ejecuta segundo
Esto se ejecuta tercero
```

Para poder comenzar a escribir los primeros programas veremos algunos elementos básicos de Python.

Los programas deben comunicarse con el mundo exterior, ya sea para obtener los datos con los que trabajarán, para entregar los resultados obtenidos, o ambas cosas.

En este curso la comunicación de nuestros programas con el mundo exterior se realizará casi exclusivamente con el usuario; tanto sea para obtener datos de entrada, como para comunicarle resultados parciales o finales.

Entonces, el único posible ingreso de datos (desde el punto de vista del programa; es decir, datos que el usuario le informa al programa) se llevará a cabo a través del **teclado** (dispositivo de entrada por defecto). Así como el único medio de salida de información (siempre desde el punto de vista del programa; es decir, el medio por el cual el programa muestra resultados o mensajes al usuario) será la **pantalla** (dispositivo estándar de salida).

1.2. Funciones

Python dispone de herramientas tanto para poder hacer la lectura desde teclado (lo que el usuario **tipea** es enviado al programa), como para poder enviar resultados a la pantalla (para que el usuario pueda leerlos).

Estas dos operaciones se llevan a cabo con **funciones**. Las **funciones** en Python siempre se usan escribiendo su nombre y a continuación, entre paréntesis, los **parámetros** o **argumentos**; es decir, los datos que enviamos para que pueda realizar su tarea.

Para leer, o ingresar, o permitir que el usuario ingrese datos, usaremos la función **input()**.

Para mostrar toda la información que el programador considere debe mostrársele al usuario, emplearemos la función **print()**.

⚠️ **Atención!**

Es muy importante respetar mayúsculas y minúsculas! PRINT() Input() o prINT() no serán reconocidas. Esto aplica para todo lo que escribamos en nuestros programas.

1.3. Variables y Constantes

Necesitamos **print**() e **input**() para realizar acciones, es decir, nos permiten escribir el **algoritmo** del programa; pero, no haremos nada sin considerar los **datos**.

Debemos tener presente que, todos los datos que nos ingresen o que el programa produzca y tengan que ser recordados o utilizados más adelante obligatoriamente serán alojados en la **Memoria Interna** (**MI**) y para hacerlo tendremos dos opciones:

Si el dato permanecerá inalterable, no cambiará de valor, entonces diremos que ese dato será una **constante**. Para referenciar a una constante en un programa simplemente la llamamos por su valor.

Ejemplo:

`2`	Es una constante numérica; y es el número **2**
`'Hola'`	Es un texto constante, la palabra **Hola**

Si por el contrario, un dato puede cambiar o *variar* su valor, será una **variable**.

Para poder manipular datos variables utilizaremos un **nombre** (o etiqueta) que le pondremos al mismo para poder identificarlo y emplearlo. El **nombre** puede ser cualquier combinación de letras y números (tengamos presente que mayúsculas, minúsculas y acentuadas son letras diferentes).

Nota:

No se puede emplear el mismo nombre para dos datos diferentes; una variable puede referenciar un sólo dato por vez. *Por lo tanto si uso un mismo nombre para un nuevo dato se pierde el valor anterior (en la jerga de programación decimos que* ***se pisa*** *el dato anterior).*

1.4. Buenas Prácticas de Programación

✔ **Sobre convención de nombres:** Para nombres de variables y funciones, se usa `snake_case`, que es básicamente dejar todas las palabras en minúscula y unirlas con un guión bajo. Ejemplos: `numero_positivo`, `sumar_cinco`, `pedir_numero`, etc.
Siempre emplear un nombre mnemotécnico, es decir que nos remita al significado que tendrá ese dato, siempre en snakecase:
`numero`, `letra`, `letra2`, `edad`

Variables

Las variables son cosas. Entonces sus nombres son sustantivos: `nombre`, `numero`, `suma`, `resta`, `resultado`, `respuesta_usuario`. La única excepción son las variables booleanas (ya las vamos a ver, son aquellas que pueden guardar dos posibles valores: verdadero o falso), que suelen tener nombres como `es_par`, `es_cero`, `es_entero`, porque su valor es true o false.

A veces es útil alguna frase para identificar mejor el contenido:
`edad_mayor_hijo`, `apellido_conyuge`

Funciones

Las funciones hacen algo. Entonces sus nombres son verbos en infinitivo. Se usan siempre verbos en infinitivo (terminan en -ar, -er, -i):

`calcul`**`ar`**`_suma`, `imprim`**`ir`**`_mensaje`, `corr`**`er`**`_prueba`, `obten`**`er`**`_triplicado`, `etc`.

De nuevo, las excepciones son las funciones que devuelven un valor booleano (V o F). Esas pueden llamarse como: `es_par`, `da_cero`, `tiene_letra_a`, porque devuelven verdadero o falso, y eso nos confirma o niega la afirmación que hace el nombre.

✔ **Sobre ordenamiento de código:** Cuando uno corre Python, lo que hace el lenguaje es leer línea a línea nuestro código. Lo que se puede ejecutar, lo ejecuta. Las funciones las guarda en memoria para poder usarlas luego.

Entonces es más ordenado y prolijo primero poner todas las funciones, y después el código "ejecutable" (si van a dejar código suelto en el archivo).

2. Tipos de Datos

2.1. Datos Simples

Los programas trabajan con diversidad de datos, no necesariamente de la misma naturaleza. A veces necesitan realizar transformaciones a números, otras veces a texto o incluso consultar si un dato es verdadero o falso. Es decir, caben diferentes tipos de operaciones a los datos a los efectos de generar resultados y nueva información. Pero, claro, dependiendo de la naturaleza o **tipo** de información cabrá la posibilidad de realizar distintas transformaciones aplicando **operadores**. Por eso, a la hora de representar información no sólo importará que identifiquemos el dato relevante y podamos conocer su valor al ejecutar el programa, sino también resultará crucial tener en claro qué tipo de tratamiento le daremos a ese dato en particular. Esto último determinará el **tipo de dato** que necesitamos.

Todos los Lenguajes disponen de un conjunto de **tipos predefinidos** de datos. ¿Qué significa que sean **predefinidos**? Que ya los conoce. Sabe cómo guardarlos en memoria y qué transformaciones puede aplicarles (operadores y contextos válidos).

Si uno simplemente escribe las siguientes líneas de código:

```
a=3
b=a*2
```

Python no tendrá dificultad en reconocer que tanto **a** como **b** son tipos de datos que ya conoce: **numéricos enteros**. Y por lo tanto reconoce cómo guardarlos en memoria, sabe que es válida la operación a*2 y puede calcularla.

Completaremos la lista:

Tipo de dato	**Ejemplos de constantes**
entero – **int**	9 -5 37 0
real - **float**	0.01 88.72156333 -12.0 0.0
complejo - **complex**	(10,9j) la componente con j del par indica la parte imaginaria
lógico - **bool**	True False (verdadero y falso)
texto - **str**	'Ana y sus HERMANAS!, 7 en total' 'APRENDO A programar ****' 'u' ''

El **texto**, **string** o **cadena** (tipo **str**), en realidad tiene dos caras. Es un dato simple (un texto), pero también es una colección de datos (caracteres). Por eso podremos realizar algunas consultas y selecciones de parte de un texto que no podemos hacer con los otros tipos de datos predefinidos. El tipo **str** de Python permite trabajar

con textos de 0, 1 o más caracteres. Debido a eso, en este lenguaje no diferenciamos mucho un **caracter** de un **texto**; ya que manipulamos un **caracter** como un **texto** de un único elemento.

La naturaleza de un texto es diferente a la de un número. Entonces, además de almacenarse internamente de manera distinta, trabajamos una **string** con sus propias reglas. Claramente no le haremos transformaciones matemáticas. Pero hay muchas transformaciones que se pueden y se necesitan realizar a textos. En general lo llamamos **Edición de Texto.** Las **cadenas** son un **tipo de Secuencia.** Una **secuencia** en Python es un conjunto contiguo de elementos con una organización interna. Cada carácter ocupa una posición determinada.

El texto: `'hola'` no es igual a `'aloh'` y tampoco a `'Holá'`

2.2. Operadores

La pregunta del millón, entonces: ¿Cuáles son los caracteres que podemos colocar dentro de una cadena? Podemos usar letras, dígitos numéricos, símbolos de puntuación, operadores, etc. Un **string** permite cualquier combinación de caracteres **Unicode** (una codificación estándar que admite una cantidad y variedad importante de caracteres). Las **constantes string** se escriben encerradas entre comillas simples o dobles. Podemos elegir cualquiera, pero no mezclar en la misma cadena. *Si abrimos con comilla simple, cerramos con ella y lo mismo para las dobles.*

Finalmente diremos que para transformar datos numéricos emplearemos **operaciones válidas** en Python, que en este caso serán coincidentes con **operaciones matemáticas** y de **edición de texto**.

A continuación una lista de **operadores aritméticos** válidos de Python:

Símbolo	Definición	Ejemplo
+	Suma	`10+cant`
-	Resta	`saldo-pago`
*****	Producto	`a*20`
/	División com Precisión Decimal	`a/3.5`
//	División Entera	`a//12`
%	Resto	`num_par%2`
+=	Suma abreviada, agrega	`a=a+3 ≈ a+=3`
-=	Resta abreviada, quita	`a=a-3 ≈ a-=3`
***=**	Producto abreviado	`a=a*3 ≈ a*=3`
/=	División abreviada	`a=a/3 ≈ a/=3`
//=	División entera abreviada	`a=a//3 ≈ a//=3`
%=	Resto abreviado	`a=a%3 ≈ a%=3`

Para alterar cualquier precedencia debemos usar (), como en matemáticas

El orden de prioridad de ejecución va a ser:

1. **paréntesis ()**
2. ***, / , // , %**
3. **+, -**

Operadores de **edición de texto** válidos:

Símbolo	Definición	Ejemplo
+	Concatenación	`'hola'+' juan'-> hola Juan`
*****	Repetición de Texto	`'ja'*3 -> jajaja`
[k] o [-k]	Selección de caracter	`a='hola'  a[0] -> h` `a[2] -> l`
[i : j : p]	Selección de una porción del texto. Siendo: i : inicio, j : final, p : pasos	`a[0:2]-> ho`
+=	Concatenación abreviada	`a = hola` `a+='y chau' -> holay chau`
***=**	Repetición abreviada	`a*=2 -> holahola`

Para alterar cualquier precedencia debemos usar (), como en matemáticas

El orden de prioridad de ejecución va a ser:

4. **paréntesis ()**
5. **corchetes []**
6. **+, ***

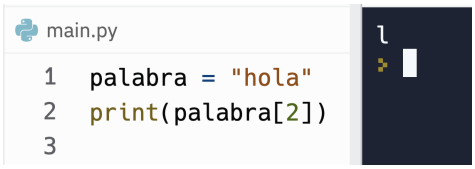
2.2.1. Manipulando Strings

Si bien esto lo vamos a ahondar en la siguiente Sesión de la materia, es importante saber que los strings, como dijimos más arriba, son un conjunto de caracteres. Pero no sólo un conjunto, sino un **conjunto ordenado**. Cada caracter tiene una posición dentro de la palabra o string y no es lo mismo, por ejemplo: CASA que SACA.

Cada caracter tiene una posición dentro del string, y las posiciones comienzan en cero, de izquierda a derecha:

```
0 1 2 3
h o l a
```

Por lo tanto, para la palabra “hola”, si quiero obtener el caracter 2, me devuelve la letra “L”:



Pero no sólo puedo obtener los caracteres en las posiciones de la palabra, sino que puedo obtener *slices* o *porciones* de la palabra, usando algo que vemos por primera vez: **rangos**.

Un rango tiene 3 partes:

```
[start : end : step]
```

- **start** es desde dónde queremos que comience nuestra porción de la palabra
- **end** es hasta dónde queremos que llegue (ojo, no incluye a la misma posición)
- **step** es si queremos que se saltee posiciones (por ejemplo, que vaya de 2 en 2. Si no aclaramos un step, va de 1 en 1.

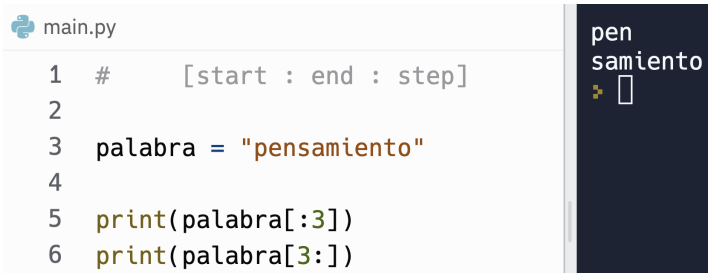
Vamos a dar un ejemplo:

Sabiendo que los caracteres comienzan en la posición 0, la posición 1 va a ser la primera letra "**e".** La 2 va a ser la **"n",** y la 3 la "s". Pero, como ya dijimos, el **end es excluyente, por lo que sólo nos quedamos con las letras de la posición 2 y 3.**

Probemos con otro rango:

En este caso, el caracter de la posición 3 es la **"s"**, y llegamos hasta la letra **"m"** en la posición 7, sin incluirlo.

Si no aclaramos inicio, se considera que comienza desde el cero. De la misma forma si no aclaramos final, se considera que finalizamos en el extremo derecho de la palabra:



Como ejemplo de los steps, podemos pedirle del caracter 0 al 10, de a 2 pasos:

¿Se animan a pensar qué se imprimiría si pedimos **palabra[::3]** ? Se lo dejamos para que prueben por su cuenta.

No se preocupen si parece difícil, de este tema vamos a ver mucho más en la próxima guía.

2.3. Input y casteos

Como ya vimos en el apunte de la Sesión 1, tenemos una forma de imprimir la edad de Juan:

```
print('La edad de Juan es 30')
```

Pero sabemos que **Juan** se irá poniendo viejo. Mejor si le preguntamos al usuario cuántos años tiene **Juan** en ese momento.

✏️ **Nota:**

Algunos ejemplos son extremadamente tontos, lo sé, pero seamos tolerantes a fin de aprender las variantes.

Entonces debemos preguntarle al usuario la **edad** de **Juan** y mostrar su respuesta.

Así que debemos hacer que el usuario ingrese un dato. No sabemos su valor, por lo tanto deberá ser colocado en una **variable** y recién allí mostraremos el resultado.

A la izquierda veremos la salida o ejecución del programa, asumiendo que el usuario ingresa “25”

```
print('Ingrese la edad de Juan')
edad = input()
print('La edad de Juan es', edad)
```

```
Ingrese la edad de Juan
25
La edad de Juan es 25
```

En la segunda sentencia **print** vemos que le enviamos 2 parámetros, separados por coma. Y es que podemos pedirle que saque la cantidad de datos que necesitemos, si son más de uno, los separamos por coma. También podemos emplear un **print**() sin argumentos; en ese caso se agrega una línea en blanco en la pantalla.

En este ejemplo le pedimos que muestre una constante de texto (por eso está entre comillas) y el valor de una variable (**edad**). Como **edad** es el nombre de una variable, no va entre comillas, y lo que se muestra es su contenido.

Para el ingreso se usó:

`edad=input()`	En esta sentencia se le pide al usuario un ingreso (input). Como en el renglón anterior se le explicó lo que se espera, sabe que debe ingresar la edad de Juan. Se recibe el dato, que es 25, en este caso, y es guardado en la variable edad También usamos una operación de almacenamiento en memoria (asignación =)

input() le pide al usuario que ingrese un dato (¿Cómo pide? Muestra en pantalla el cursor titilando). ¿Y qué hacemos con ese número? Se guarda o almacena en la variable **edad**.

✏️ **Nota:**

El input se quedará esperando a que el usuario (nosotros) ingresemos algo, hasta que apretemos Enter.

La que empleamos fue una operación de **asignación**, siendo **=** el operador de asignación en Python. La operación **asignación** se ejecuta de la siguiente manera: se evalúa la expresión a la derecha del igual y el resultado se guarda en la variable a la izquierda del igual (para ser más precisos deberíamos decir que **se pone a la etiqueta de la izquierda del igual a apuntar al valor resultante de evaluar la expresión de la derecha**).

✏️ **Nota:**

¡La operación asignación no es conmutativa! Siempre se escribe el nombre de la variable a la izquierda del igual y la expresión que producirá el valor resultante a la derecha.

Algunos ejemplos con asignaciones:

`digito = '2'`	Guarda el caracter **2** en la variable **digito**
`saludo = 'hola'`	Guarda el texto **hola** en la variable **saludo**
`numero = 3 * 2`	Guarda el resultado de multiplicar **3** por **2** en la variable **numero**
`n = n + 0 - 2.15`	Guarda el resultado de sumarle **0** al valor corriente de **n** y restarle **2.15** en la variable **n** (acá **pisa** el valor anterior de **n**)

Cuando se ejecuta una función **input**() muchas veces el usuario verá que el programa espera que ingrese un dato, pero no necesariamente sabe cuál es la información que aguarda. Para aclararle al usuario lo que esperamos que ingrese, colocamos el **print**() explicativo antes del **input**().

Los ingresos se pueden hacer en combo **print-input**. O podemos emplear un argumento opcional de la función **input**() que admite se le envíe un texto. Este será mostrado por pantalla antes de quedar a la espera del ingreso, y por lo tanto, podemos usarlo para explicarle al usuario qué información estamos aguardando.

¿Cómo podríamos reescribir el programa anterior empleando el argumento de **input**()?

```
edad=input('Ingrese la edad de Juan: ')
print('La edad de Juan es', edad)
```

Ahora supongamos que queremos extender el programa para preguntar la edad de cualquier persona.

Líneas de Código	**Ejecución**

`print('quién?')`
`nombre = input()`
`print()`
`edad = input(f'Ingrese la edad de {nombre} ')`
`print('La edad de', nombre,'es',edad)`
	`quién?`
	`Ana`
	`Ingrese la edad de Ana 17`
	`La edad de Ana es 17`
	`>>>`

En este caso tenemos un problema…

No sabemos qué nombre ingresará el usuario, así que no podemos colocar el nombre dentro del cartel (como lo hicimos con Juan), como podemos observar, el texto que le enviamos al **input**() como argumento es constante. ¿Cómo introducimos un dato variable en un texto constante? Hay muchas maneras de hacer esto pero no es buena idea abundar en múltiples recetas, sobre todo cuando se está aprendiendo a cocinar. Voy con una propuesta muy simple: cuando queramos introducir datos que van a cambiar (variables) dentro de

un texto constante podemos usar la notación de **cadena de formateo**. Entonces colocamos una **f** (puede ser **f** o **F**), a continuación el texto encerrado entre comillas (texto constante) y en el lugar en que queramos que aparezca el dato variable colocamos el nombre de la variable encerrado entre **{}**. Incluso podemos colocar más de un dato variable dentro de un texto.

✏️ Nota: Imprimiendo Strings y Variables

El formato de un print es el siguiente: **print paréntesis contenido paréntesis**

El contenido puede ser:
- una variable
- un texto
- ambas, mezcladas.

Ejemplo:
```
dia_semana = "Miércoles"
dia_mes = 8
print(dia_semana) # -> Esto nos imprime: "Miércoles"
print(dia_mes) # -> Esto nos imprime: 8
```

También puedo mezclarlo de distintas formas. Estas son: usando +, usando comas, usando f"".

Veamos los prints anteriores usándolos en ejemplos:
```
# Usando +:
print("El día de la semana es: " + dia_semana)
print("El día del mes es: " + str(dia_mes))

# Usando comas:
print("El día de la semana es: ", dia_semana)
print("El día del mes es: ", str(dia_mes))

# Usando f""
print(f"El día de la semana es: {dia_semana}")
print(f"El día del mes es: {str(dia_mes)}")
```

Son todas formas iguales de imprimir lo mismo. Pero, independientemente de cuál usemos, el formato siempre es el mismo: `print paréntesis contenido paréntesis`

El contenido es el que cambia, dependiendo de cómo concatenamos lo que queremos mostrar: usando `+`, `comas` o `f""`

Ejercitación

Vamos a calcular la edad de un hermano mayor.

Pediremos nombre y edad de una persona, luego cuántos años más tiene su hermano y su nombre e informaremos la edad del hermano.

Debemos pedir el nombre de las dos personas. Debemos pedir la diferencia de edad. ¿Tendremos que pedir las dos edades? En realidad, sólo una, la otra podemos calcularla. Luego decidimos qué datos presentar como salida de ¡nuestro maravilloso programa!

Nuestro programa:

```
nombre_menor=input('quién? ')
edad_menor=int(input(f'Ingresá la edad de { nombre_menor } '))
nombre_mayor=input(f'Cómo se llama el hermano mayor de {nombre_menor}? ')
diferen=int(input(f'Cuántos años más tiene {nombre_mayor}?  '))
edad_mayor = edad_menor + diferen
print(nombre_menor,'tiene',edad_menor,'años')
print(nombre_mayor,'es mayor y tiene', edad_mayor, 'años')
```

Líneas de Código	Ejecución

main.py

```
1  nomMen=input('quién? ')
2  edadMen=int(input(f'Ingresá la edad de { nomMen } '))
3
4  nomMay=input(f'Cómo se llama el hermano mayor de {nomMen}? ')
5  diferen=int(input(f'Cuántos años más tiene {nomMay}?  '))
6
7  edadMay=edadMen+diferen
8
9  print(nomMen,'tiene',edadMen,'años')
10 print(nomMay,'es mayor y tiene', edadMay, 'años')
11
```

```
quién? Ana
Ingresá la edad de Ana 17
Cómo se llama el hermano mayor de Ana? Luis
Cuántos años más tiene Luis?  3
Ana tiene 17 años
Luis es mayor y tiene 20 años
>
```

Acá aparece la función **int**(). ¿Para qué se usa? Sea lo que sea que ingrese el usuario por teclado, la función **input**() se lo envía al programa como un objeto de texto. Muchas veces necesitamos que los números **no sólo se vean como números, sino que se comporten como tales**, por ejemplo para aplicarles una operación aritmética (como en este caso en que debo sumarle el número **diferen** al número **edad_menor**).

Por lo tanto el resultado del **input**() (siempre devuelve texto) en ese caso debe ser convertido a un objeto numérico, eso se hace con una **función de casteo int**(), que es una función que convierte y devuelve el dato que se le envía entre paréntesis como número entero. Por supuesto que el dato enviado debe ser compatible con un valor numérico entero. Si lo que recibe **int**() es un número con precisión decimal devuelve sólo su parte entera.

En el programa le enviamos a **int**() el resultado del ingreso del usuario; el *texto* **17**, por ejemplo. **int**() devuelve el *número* **17.** A un *número* **(17)** le podemos sumar la diferencia **(3),** que será un *número***,** ya que, si nos fijamos, a ese ingreso se le aplicó **int**() también.

¡Un tip más! Cuando construimos programas complejos y/o extensos es difícil seguirlos y entenderlos bien para cambiarlos. Para facilitarnos la tarea, los programadores dejamos pistas y explicaciones en el programa fuente, que sólo serán leídas por programadores (Python las descarta cuando traduce el programa para ejecutar). Estas ayudas que dejamos en el camino se llaman **comentarios**.

En Python podemos comentar una línea empleando el símbolo **#**. De ese modo, todo lo que esté escrito a la derecha del símbolo en la misma línea será ignorado en la ejecución. Si tenemos que explicar mucho y vamos a utilizar varias líneas, en lugar de usar un **#** al inicio por cada línea de comentario sólo empleamos triple comilla (otra vez pueden ser simples o dobles, pero abrimos y cerramos con lo mismo) al comenzar el comentario; y podríamos dedicarnos a escribir nuestra primera novela ¡si quisiéramos! Sólo debemos recordar cerrarlo con otro juego de triple comilla.

Ejemplitos:

```
print('Hola a todos')  #saludo por pantalla
#el programa realiza el cálculo de la superficie de un triángulo
''' la función realiza suma de argumentos
Admite cantidad variable de argumentos
Devuelve la suma  '''
```

3. Funciones

Emplear **funciones** predefinidas o escritas por terceros es de gran ayuda. Imaginemos que cada vez que quisiéramos mostrar algo por pantalla tuviéramos que programar todo lo que la función **print**() hace. Verificar el estado del dispositivo, la posición actual del cursor, colocar los datos en la pantalla respetando los límites de filas y columnas, saturar con blancos cuando sobra espacio, ¡puff! No es trabajo para impacientes… Ahora, simplemente hagamos un ejercicio intelectual y visualicemos la cantidad de veces que mostramos por pantalla carteles, datos, resultados, etc… ¡Si, exacto! Mil veces lo mismo, o al menos parecido. Tiene razón Python en definir una función que hace el trabajo de mostrar datos en pantalla de forma genérica y los programadores sólo la usamos enviándole parámetros o argumentos para ajustar su funcionamiento a nuestras propias necesidades.

3.1. Funciones Predefinidas

Disponemos de una amplia gama de funciones **predefinidas** (que Python ya conoce), tales como **print**(), **input**(), **abs**(), **round**(), **int**(), **float**(), **str**(), **bool**(), **len**(). Hay muchas, pero seguro que siempre necesitaremos más. Cuando escribimos nuestros programas normalmente identificamos partes del mismo que podemos escribir separadamente como una **función** genérica. Esto lo hacemos porque nos damos cuenta que podremos reutilizar ese código en varias partes del programa o porque, siguiendo el lema: *Divide y vencerás,* organizamos el programa sustituyendo partes por llamadas a una función y generamos más claridad. En cualquier caso, Python nos permite trabajar con nuestras **propias funciones**. Ahora claro, una cosa es que

Python sea tolerante y otra muy distinta es que ¡pierda el control! Todo objeto (datos o programas) que Python vaya a utilizar debe ser conocido previamente por él. Y básicamente tiene dos maneras de conocer un objeto:

- ✔ Está predefinido
- ✔ Debemos presentárselo

Cuando trabajamos con funciones que no son **predefinidas** antes de usarlas (**invocarlas**) debemos hacer que Python las conozca. Este proceso se llama **definición**.

3.2. Firma de la Función

Para definir una función propia en Python es necesario escribir una línea de **encabezado** y varias líneas de **cuerpo**. La línea de **encabezado** deberá comenzar con la palabra **def** seguida del nombre que le pondremos a nuestra función (podemos aplicar las mismas reglas de nombre para las variables) y luego paréntesis para terminar con **dos puntos.** A esta línea se le llama **la firma de la función**.

Dentro de los paréntesis, si los necesitamos, indicaremos los nombres de todos los **parámetros** separados por coma. Los **parámetros** serán las variables en las que se van a copiar los datos que se enviarán a la función (cuando sea invocada) para que pueda ejecutarse. Cuando terminamos la línea de encabezado, comenzamos a escribir **qué hará** nuestra función igual que en un programa cualquiera (eso es el **cuerpo** de la función). Sólo que, para que Python sepa dónde acaba ese **cuerpo**, debemos desplazarlo hacia la derecha un poco y cada línea del **cuerpo** deberá comenzar a la misma altura. Este desplazamiento a derecha se conoce como **indentación**. Todas las sentencias que tienen una o varias líneas de **encabezado**, llamadas **Sentencias Estructuradas**, tienen su **cuerpo indentado**.

3.3. Devolviendo un Resultado

Si queremos que la función devuelva un resultado será necesario elegir uno de los datos que maneja la función (**sólo uno**) y devolverlo a quién invocó a la función, empleando la sentencia **return**. Escribiremos **return** y el dato.

💡**¡Atención!**

Una sentencia return no sólo devuelve un dato, también determina la finalización de la ejecución de la función. Si no se coloca ningún return, la función no devuelve resultado y acaba su ejecución cuando el FCP termina de ejecutar el cuerpo.

Las funciones en programación trabajan como las funciones matemáticas:

`f(x) = y`

Sabemos que para el mismo **x** siempre se obtiene el mismo **y**. O sea que para iguales valores de entrada, el mismo resultado. También podemos obtener el mismo resultado para diferentes combinaciones de valores de entrada (eso pasa también con algunas funciones matemáticas).

Y en matemáticas, como en programación, las funciones se emplean para clarificar resoluciones y generalizar operaciones.

También podríamos asimilar el trabajo de programación empleando funciones como una fábrica que terceriza algunos procesos. Entrega ciertos materiales y recibe una pieza terminada. Idealmente no debe preocuparse de cómo se llevará a cabo eso y además, puede enviar a producir distintas piezas con diferentes proveedores.

Igualmente cuando escribimos un programa empleando funciones, si están correctamente diseñadas, no deberíamos preocuparnos de cómo se resuelven las tareas dentro de las funciones ni preocuparnos por los detalles (esa característica o propiedad se llama **encapsulamiento** y es muy interesante a nivel de diseño). Sólo que si las funciones que vamos a emplear serán de nuestra autoría deberemos comportarnos como el dueño de una fábrica que terceriza trabajos en otra firma de su titularidad. En algún momento tendrá que quitarse la gorra de cliente y deberá colocarse la gorra de tercerista para completar la tarea.

3.3.1. Diferencias entre Return y Print

El **return** devuelve un valor cuando se invoca a la función.
El **print** lo manda a la pantalla, pero no devuelve nada.

Para el return, cuando se invoca a la función, se tiene que atajar (asignar) el valor que devuelve.
Para el print no hace falta.

Ejemplo print:

```
def imprimir_saludo(nombre):
  print("Hola,", nombre)

nombre = "Lucas"
imprimir_saludo(nombre)
```

Ejemplo return:

```
def obtener_saludo(nombre):
  return "Hola, " + nombre

nombre = "Lucas"
saludo = obtener_saludo(nombre) # -> Esto es lo importante, antes no se asignaba ningún valor, ahora que retorna algo, sí
```

Si no hacés nada más, no vas a ver nada en pantalla. Entonces lo printeamos para ver que esté todo ok:

```
print(saludo)
```

3.4. Invocando Funciones

Ya dijimos que Python es bueno pero no ingenuo. Y que antes de poder aplicar una función (momento de la **invocación** o **llamada**) debe saber de qué se trata. Por lo tanto tenemos tiempo hasta justo la sentencia anterior a la **invocación** para **definir** una **función**. Sin embargo, **es una buena práctica (importante) de programación definir todas las funciones propias al principio del programa, una a continuación de otra.** Esta práctica sencilla permite mejorar la comprensión de los programas y también generaliza la definición para que la función pueda ser invocada no sólo desde el programa (lo llamaremos **Programa Principal**, porque las funciones son mini programitas) sino también desde el cuerpo de otras funciones.

💡**¡Atención!**

Si escribimos nuestros programas siguiendo estas reglas, deberemos comenzar a leerlos a partir de la primera línea del Programa Principal, que quedará escrito al final de todo.

¿Cómo funciona un programa que invoca funciones? Igual que cuando invocamos **print**() o **input**(). El **FCP** comienza desde la primera sentencia del Programa Principal y va ejecutando cada sentencia. Cuando encuentra una llamada, el control salta a la **primera sentencia** del cuerpo de la función. El cuerpo será completamente ejecutado (o hasta encontrar un **return**) y entonces el **FCP** volverá al lugar desde dónde fue **invocada** la función y proseguirá la ejecución desde allí.

Cuando sucede una **invocación** pasan varias cosas internamente, pero lo más importante a tener en cuenta es que antes de que el **FCP** pegue el salto al cuerpo de la función, sucede una operación que se denomina **Matching** o **Apareo de Parámetros**. Python toma cada **argumento** que se le pasa a la función en la **llamada** y lo copia en la variable que está esperando en esa posición en la línea del encabezado de la función. De ese modo trabajará como un tercero, con copia de los datos y algunos datos propios, sin molestar ni alterar datos del programa o la función llamadora. Tanto los datos que una función recibe como argumentos, como los que genera por su cuenta no son accesibles desde afuera. Sólo ella los conoce y como es su dueña, cuando la función termina su ejecución (**finaliza su vida**) estos datos privados también desaparecen. La memoria que dispone una función para ser ejecutada (con sus sentencias, datos y resultados) se llama **ámbito** de la función.

Veamos un ejemplo para entender cómo funciona. Haremos un programa que permita obtener un cálculo (Puede ser cualquier operación aritmética binaria válida) entre dos números:

```
#Cálculo de operación aritmética binaria
print('Cálculo de Operación')
num1=input('Num 1: ')
```

```
num2=input('Num 2: ')
op=input('Operador (+ - *  / // % ): ')
result=eval(num 1+op+num2)
print(num1,op,num2,'=',result,sep='')
```

✏️ **Nota:**

La función eval() permite obtener el resultado de evaluar el texto que se le pasa como argumento, siempre que el mismo sea compatible con una expresión aritmética válida.

Una ejecución posible sería:

```
#Cálculo de Operación
Num 1: 112
Num 2: 33
Operador (+ - *  / // % ): %
112%33=13
>>>
```

Ahora escribiremos una versión empleando funciones:

```
def calculo (n1,n2,op):
    return eval(n1+op+n2)

#Cálculo de operación aritmética binaria
print('Cálculo de Operación')
num1 = input('Num 1: ')
num2 = input('Num 2: ')
op = input('Operador (+ - *  / // % ): ')
result = calculo(num1,num2,op)
print(num1,op,num2,'=',result,sep='')
```

Esta versión produce el mismo resultado que la anterior. Para el usuario es imposible determinar si el programa fue escrito de una manera u otra. *Pero las cosas adentro funcionan distinto*... Como podemos observar los nombres de los argumentos cuando se invoca y en la definición pueden ser los mismos o distintos. No hay problemas con eso, lo único importante es el orden. Recordemos que el tercerista recibe lo

que le envían y se mete puertas adentro a resolver el problema sin molestar al cliente, ni pedirle nada más. Cuando termina le devuelve el resultado y ya.

💡**¡Atención!**

Por si no quedó claro aún. Cuando se invoca se debe enviar valores que serán el resultado de evaluar una expresión del tipo requerido. Una constante, el contenido de una variable, el resultado de una función, una operación son expresiones válidas. Una vez que se obtiene su valor, se pasa a la función. POR CONTRAPOSICIÓN la función sólo puede tener variables como argumentos. Ya que define los parámetros cuando necesita RECIBIR algún valor (que no conoce) desde afuera, así que será si o si una variable.

En este apunte vimos funciones, variables, constantes, buenas prácticas de programación, datos simples con sus operadores, y casteos. Si quedan dudas, recuerden que pueden consultarnos por el foro del campus. Si es necesario, pueden agregar algún ejemplo, o imágen, que facilite la interpretación de la consulta.

4. Bonus Track:

Más funciones Predefinidas de Python

Función	Definición	Ejemplo de Uso
print()	Imprime un mensaje o valor en la consola	print("Hello, world!")
input()	Lee una entrada de texto desde el usuario	name = input("Enter your name: ")
abs()	Devuelve el valor absoluto de un número	abs(-5)
round()	Redondea un número al entero más cercano	round(3.7)
int()	Convierte un valor en un entero	x = int("5")
float()	Convierte un valor en un número de punto flotante	y = float("3.14")
str()	Convierte un valor en una cadena de texto	message = str(42)
bool()	Convierte un valor en un booleano	is_valid = bool(1)
len()	Devuelve la longitud (número de elementos) de un objeto	length = len("Hello")
max()	Devuelve el valor máximo entre varios elementos o una secuencia	max(4, 9, 2)

min()	Devuelve el valor mínimo entre varios elementos o una secuencia	min(4, 9, 2)
pow()	Calcula la potencia de un número	result = pow(2, 3)
range()	Genera una secuencia de números	numbers = range(1, 5)
type()	Devuelve el tipo de un objeto	data_type = type("Hello")
round()	Redondea un número a un número de decimales específico	rounded_num = round(3.14159, 2)
isinstance()	Verifica si un objeto es una instancia de una clase específica	is_instance = isinstance(5, int)
replace()	Reemplaza todas las apariciones de un substring por otro	text = "Hello, World!" new_text = text.replace("Hello", "Hi")

Bibliografía Adicional

Wachenchauzer, R. (2018). *Aprendiendo a programar usando Python como herramienta*. Capítulos 1,2,3,6