# Curso de Python para Analisis de Datos

## Clase 3

**Indice:**



## MANIPULACIÓN DE VARIABLES

En Python, toda variable se considera un objeto. Sobre cada objeto, pueden realizarse diferentes tipos de acciones denominadas métodos. Los métodos son funciones pero que se desprenden de una variable. Por ello, se accede a estas funciones mediante la sintaxis:

`variable.funcion()`

En algunos casos, estos métodos (funciones de un objeto), aceptarán parámetros como cualquier otra función.

`variable.funcion(parametro)`

## Funciones

Las funciones nos permiten agrupar una o más líneas de código bajo un mismo nombre. Su objetivo principal es el de evitar la repetición de código, haciendo de un archivo de código de fuente más claro, legible y fácil de mantener.

In [ ]:
```
# Por ejemplo consideremos este programa:
print("¡Hola mundo!")
print("Desde Python.")
a = 5
b = 7
c = a * b
print(c)
print("¡Hola mundo!")
print("Desde Python.")
```

Es algo muy común que algunas porciones de código se repitan en distintas partes de un programa, como es el caso aquí de las primeras y últimas dos líneas. Para evitar esto, lo que se hace es darle un nombre a las líneas repetidas creando así una `función`. La sintaxis, siguiendo el ejemplo anterior, es la siguiente:

In [ ]:
```
def mi_funcion():
    print("¡Hola mundo!")
    print("Desde Python.")
```

La creación de una función requiere de la palabra reservada **`def`** , seguida de un nombre a elección, un par de paréntesis (que más adelante veremos para qué sirven) y dos puntos. El bloque de código dentro de la función debe tener sangría, tal como ocurría con los bucles y los condicionales.

Ahora que tenemos esta función, en los lugares donde hacíamos uso de esas dos líneas simplemente pondremos `mi_funcion()` .

In [ ]:
```
def mi_funcion():
    print("¡Hola mundo!")
    print("Desde Python.")


mi_funcion()

a = 5
b = 7
c = a * b
print(c)

mi_funcion()
```

En todos los lugares donde aparezca el nombre de una `función` junto a un par de paréntesis, Python ejecutará las líneas de código que se encuentran dentro de ella. A esto lo denominamos llamar (o **invocar**) a una función .

In [ ]:
```
# Si modificamos el código de nuestra función de esta manera:
def mi_funcion():
    print("¡Hola mundo!")
    print("Desde Python.")
    print("Y Geany.")

#Veremos que en ambas llamadas a mi_funcion() , ahora se ejecutan tres líneas
en lugar de dos.
```

# Argumentos

Cambiemos el ejemplo y consideremos ahora el siguiente código:

In [ ]:
```
alumnos = ["Pablo", "Juan", "Matias", "Martin"]
notas = [5.5, 9, 6.25, 8]

print("Alumnos:")
for alumno in alumnos:
    print(alumno)

print("Notas:")
for nota in notas:
    print(nota)
```

En este caso también tenemos cierta repitencia: los bucles `for` son muy similares, aunque cambia el nombre de la `lista` sobre la que se aplican. No obstante, el método es siempre el mismo: recorrer cada uno de los elementos de la lista e imprimirlos en pantalla vía `print()` **(a partir de ahora, cuando indicamos paréntesis luego del nombre de una instrucción en este material, querremos decir que se trata de una función).**

Si tuviésemos que hacer una versión genérica de ese bucle “for”, podríamos expresarla así:

```
- for elemento in lista:
      print(elemento)
```

Donde lista puede ser alumnos o notas , o bien cualquier otra lista de la cual queramos imprimir sus elementos. Esto bien podemos hacerlo en una `función`:

```
- def imprimir_elementos(lista):
    for elemento in lista:
        print(elemento)
```

Lo que hacemos aquí es indicar entre paréntesis aquellas variables (que llevan el nombre de argumentos de una función) que serán “llenadas” por el usuario al momento de llamar a la función. Como en nuestro caso aún no sabemos cuál es la lista que se querrá imprimir, simplemente ponemos un nombre genérico lista .

In [ ]:
```
#Ahora nuestro código se vería así:

def imprimir_elementos(lista):
    for elemento in lista:
        print(elemento)

alumnos = ["Pablo", "Juan", "Matias", "Martin"]
notas = [5.5, 9, 6.25, 8]


print("Alumnos:")
imprimir_elementos(alumnos)


print("Notas:")
imprimir_elementos(notas)
```

En el código de la `función` se “reemplaza” lista por lo que indicamos entre paréntesis al llamarla; en este caso, la lista alumnos . Del mismo modo ocurre en el segundo caso para `imprimir_elementos(notas)`.

Incluso podemos usar dicha `función` para imprimir cualquier `lista`, aunque no la hayamos guardado en una variable:

In [ ]:
```
imprimir_elementos([1, 2, 3, 4, 5])
```

Ahora bien, de nuestra función `imprimir_elementos()` decimos que requiere un **argumento**, que es el nombre de la lista. Una función puede tener múltiples argumentos, la cantidad que nosotros precisemos.

Para indicar más de uno, separamos sus nombres por comas. Por ejemplo:

In [ ]:
```
def imprimir_suma(a, b):
    print("Resultado:")
    print(a + b)


imprimir_suma(7, 5)
imprimir_suma(-5, 3.5)
```

La función imprimir_suma() requiere dos `argumentos`, al primero de ellos le dimos el nombre a y, al segundo, b . Al momento de llamarla, indicamos en ese mismo orden los valores que queremos otorgarle, también separados por comas. (Aunque en este caso es indistinto, por cuanto en una suma el orden de los sumandos no altera el resultado).

# Valor de retorno

Las funciones pueden opcionalmente tener un resultado, así como vimos que lo tenía la expresión 7 + 5 o bien la instrucción `input` :

*nombre = input("Escribe tu nombre: ")*

En efecto, `input()` es una `función incorporada`. Es decir, una función que ya viene integrada con Python.

El resultado de una función se establece usando la palabra reservada **`return`** . De ahí que a ese resultado también se lo conozca como valor de retorno .

In [ ]:
```
#Ejemplo
def sumar(a,b):
    return a+b

resultado = sumar(7,5)
print(resultado)
print(sumar(-5, 3.5))
```

Aquí hemos creado una función `sumar()` que requiere dos argumentos y cuyo resultado es la suma de ambos.

El valor de retorno de una `función` puede ser cualquiera de los cuatro tipos de dato básicos que vimos, así como también listas. Por ejemplo, la siguiente función simula el comportamiento de la función incorporada `range()` :

In [ ]:
```
def rango(desde, hasta):
    numeros = []
    while desde < hasta:
        numeros.append(desde)
        desde = desde + 1
    return numeros


lista = rango(1, 6)
print(lista)
```

En este caso, el resultado o valor de retorno de `rango(1, 6)` es la lista [1, 2, 3, 4, 5] .

Las funciones son una herramienta fundamental para la programación, tanto en Python como en el resto de los lenguajes. Cuanto más comprendemos su concepto, más claro se torna cuán imprescindibles son. Veremos también que el lenguaje está colmado de funciones incorporadas (tales como `int()` , `range()` , `print()` , `input()` , entre muchos otros).

# Diccionarios

Otra estructura de datos muy útil son los diccionarios. Los diccionarios en Python son un tipo de estructuras de datos que permite guardar un conjunto **no ordenado** de pares **clave-valor**, siendo las claves únicas dentro de un mismo diccionario (es decir que no pueden existir dos elementos con una misma clave).

En Python, los diccionarios se definen utilizando llaves `{}` y cada uno de los elementos estan separados por comas. Cada elemento lo definimos con su par clave:valor, pudiendo ser la clave y el valor de cualquier tipo (`int`, `float`, `string`, `bool`).

Para crear un diccionario vamos a emplear llaves y separar sus miembros por comas, que an su vez serán pares de una clave y un valor separados por dos puntos. Por ejemplo:

In [ ]:
```
alumno1 = {"nombre": "Pablo", "cursos": 3}
```

Este código crea un diccionario de nombre alumno1 cuyas claves son "nombre" y "cursos" , y sus valores, respectivamente, "Pablo" y 3 . Los valores de un diccionario pueden ser de cualquier tipo de dato, incluidos otros diccionarios y listas.

Las claves, en cambio, tienen ciertas restricciones (además que **no pueden repetirse**), no obstante, por lo general serán cadenas (tal como en este ejemplo) o bien números enteros. Para luego acceder a alguno de sus valores, simplemente indicamos la clave entre corchetes:

In [ ]:
```
print(alumno1["nombre"])
print(alumno1["cursos"])
```

Del mismo modo podemos cambiar los valores:

In [ ]:
```
alumno1["cursos"] = 4
```

O bien crear nuevos pares clave-valor:

In [ ]:
```
alumno1["curso_actual"] = "Python"
```

In [ ]:
```
# Usando un bucle "for" sobre un diccionario tenemos acceso a cada una de sus claves. Por ejemplo:
for clave in alumno1:
    print(clave)
```

In [ ]:
```
#Si luego de cada clave queremos imprimir su valor, podemos hacer:
for clave in alumno1:
    print(clave)
    print(alumno1[clave])
```

Como verás, los diccionarios nos permiten crear estructuras de datos más complejas, por ejemplo, para almacenar la información de un alumno, algo que hubiese sido más engorroso de lograr usando los tipos de dato básicos o las listas.

# Tuplas

Existe otro tipo de objeto llamado **tupla**. Es muy similar a las listas con la particularidad de que es *inmutable*, es decir que no podemos modificar sus elementos una vez creado. Sintácticamente, se diferencia de las listas utilizando paréntesis en lugar de corchetes.

In [ ]:
```
# Creamos una tupla
mi_tupla = ('esto', 'es', 1, 'tupla')
```

In [ ]:
```
#Muestra el contenido de la variable mi_tupla
print(mi_tupla)
```

In [ ]:
```
#Chequea el tipo de variable de mi_lista
type(mi_tupla)
```

Ahora intentemos modificar el valor numérico 1 por la palabra 'una'

In [ ]:
```
mi_tupla[2]
```

In [ ]:
```
mi_tupla[2] = 'una'
```

Como vemos, no podemos modificar los elementos. Si queremos hacer un cambio en la tupla, debemos crearla nuevamente y sobreescribir la tupla anterior.

In [ ]:
```
mi_tupla = ('Esto', 'es', 'una', 'tupla')
```

Las tuplas comparten muchas propiedades con las listas. A continuación veremos algunas:

In [ ]:
```
#Indexación
mi_tupla[1:]
```

In [ ]:
```
#Chequear si un elemento está en la tupla
'una' in mi_tupla
```

In [ ]:
```
#Concatenación de tuplas
mi_tupla + ('Agrego', 'elementos')
```

In [ ]:
```
a = [1,2,3]
a = tuple(a)
a
```

In [ ]:
```
a = list(a)
a
```

# Ejercicios:

- Escribi una función que tome dos números enteros y calcule su división, devolviendo si la división es exacta o no.

- Escribi una función que simule un juego en el que dos jugadores tiran un dado. El que saque el valor más alto, gana. Si la puntuación coincide, empatan.

- Escribi una función que tome dos números enteros y devuelva qué número es pares y cuál impar.

- Escribi una función que tome una lista de números y calcule la suma de sus elementos.

- Escribi una función que tome dos palabras y diga si riman o no. Si coinciden las tres últimas letras debe devolver que riman. Si coinciden sólo las dos últimas tiene que decir que riman algo y si no, que no riman.

- Escribi una función para convertir temperatura de grados celsius a fahrenheit (Formula : F = C * 9/5 + 32)

- Escribi una función que devuelva la serie de Fibonacci entre 0 y 50.

**Para seguir practicando chequea este sitio web con ejercicios (https://projecteuler.net/archives).**

$c = \sqrt{a^2 + b^2}$