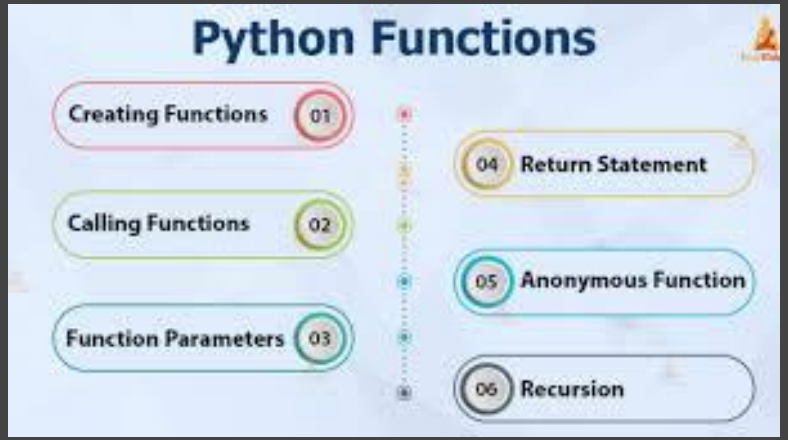
# Clase 3

Repaso de lo que vimos en la **#clase2**

CÓMO CREAR UN STRING CON VARIABLES
CODEO DE LOS EJERCICIOS

# UNIDAD VI
# FUNCIONES, MÉTODOS, FUNCIONES PREDEFINIDAS Y COMPRENSIÓN DE LISTAS




# Funciones

Una función es una *parte de un programa (subrutina) con un nombre, que puede ser invocada (llamada a ejecución) desde otras partes tantas veces como se desee, es decir, un bloque de código que puede ser ejecutado como una unidad funcional*; opcionalmente puede recibir valores, se ejecuta y puede devolver un valor. Desde el punto de vista de la organización, podemos decir que una función es algo que permite un cierto orden en el programa.


```
def nombre(parámetros):
    bloque de código...
    bloque de código...
    bloque de código…
    return valor_de_retorno
```


**nombre**: sigue las mismas reglas que los nombres de las variables

**parámetros**: no son obligatorios

**return**: no es obligatorio

# ¿Qué significa que no son obligatorios?

Que una función puede o no tener parámetros (datos de entrada) y puede o no tener un valor de retorno (datos de salida).

Pero:

- si tiene parámetros en su definición, los debe tener en su llamada
- si tiene valor de retorno, para imprimir éste, hay que imprimir la llamada a la función

```python
# Definir la función suma

def suma(a, b):
    return a + b

# llamar a la función suma
suma(1, 2)

# llamar a la función suma e imprimir su retorno
print(suma(1, 2))
```

# Parámetros por default

Muchas veces se le pasan parámetros por default a las funciones para que, si al ser llamada no tiene parámetros incluidos, no de error y simplemente se tomen como valores los de default.

```
# Definir la función suma

def suma(a=0, b=0):
    return a + b
```

# Parámetros indefinidos

Cuando no sé cuántos parámetros voy a necesitar cuando la función sea llamada, se le asigna un número indeterminado de parámetros de la siguiente manera:

```python
def suma(*args):
    sumatoria = 0
    for a in args:
        sumatoria += a
    return sumatoria


print(suma(1, 2, 3, 4, 5, 6, 7))
```

# Funciones *lambda*

Las funciones lambda *son funciones anónimas, es decir sin nombre, cuyo contenido es una única expresión en lugar de un bloque de código*.

```
lambda parámetros: acción
```

```python
suma = lambda a, b: a + b

print(suma(5, 2))
```

```python
reverse = lambda string: string[::-1]

print(reverse("hola"))
```


# Ámbitos y alcances

En Python, como en la mayoría de los lenguajes, *las variables que se utilizan pertenecen a un ámbito en particular, por lo que se dice que tienen alcance local o alcance global (scopes)*.

Dependiendo del alcance, se puede llamar o utilizar una variable desde varios sectores del programa o no.


Las funciones crean su propio espacio de nombres, el cual deja de existir tan pronto como la función invocada concluye su ejecución.

*A estos espacios de nombres diferenciados se les conoce como ámbitos y evitan que objetos definidos con nombres idénticos dentro de una función sobreescriban el espacio de nombres del intérprete*.

```python
variable = 50

def locales():
    variable = 25
    print(variable)

print(variable)
locales()
print(variable)
```

```
50
25
50
```

Para poder crear una variable global dentro de la función, es necesario utiliza la palabra reservada global en la declaración de la variable.

```python
variable = 50

def locales():
    global variable
    variable = 25
    print(variable)

print(variable)
locales()
print(variable)
```

```
50
25
25
```

# Decoradores

Los decoradores son *un tipo de función que, como su nombre lo indica, se crean para modificar (decorar) a otras funciones agregando funcionalidades extras*.

Un decorador recibe como parámetro una función y devuelve como valor de retorno otra función.

```
def función_decorador(función):

    def función_interna():
        bloque de código de func int
        ...
    return función_interna
```

Luego, para indicar que una función será decorada por la función decoradora se debe hacer lo siguiente:

```
@función_decorador
```

```
def función...
```

```python
# Creo la función decoradora

def funcion_deco(funcion_parametro):
    def funcion_interna(*args):   # agrego un valor indeterminado de parámetros kw
        print("Vamos a realizar un cálculo: ")
        funcion_parametro(*args)   # y aquí también, si no va a dar error
        print("Hemos terminado el cálculo")
    return funcion_interna  # aquí sigue sin poner paréntesis ni parámentos!

@funcion_deco  # asigno a la función suma como función_parametro
def suma(a, b, c):
    print(a + b + c)

suma(1, 2, 3)
```

Conclusión: los decoradores alteran de manera dinámica la funcionalidad de una función, método o clase sin tener que hacer subclases o cambiar el código fuente del objeto decorado. En el sentido de Python, un decorador es algo más, **incluye el patrón de diseño**, pero va más allá, los decoradores y su utilización en nuestros programas nos ayudan a hacer nuestro código más limpio y auto-documentarlo.

# Recursividad

Se denomina función recursiva a una *función que se invoca al menos una vez a sí misma*. Cada vez que una función se invoca a sí misma, Python crea un nuevo objeto de tipo function con las mismas características que la función original, pero con un ámbito totalmente nuevo y de nivel inferior a la función original.

```python
def factorial(numero):
    if numero == 1:
        return 1
    else:
        fact = numero * factorial(numero - 1)
        return fact
```

La función factorial() se invoca recursivamente y, cada vez que lo hace, el valor del argumento decrece en 1 de forma sucesiva hasta que el parámetro numero alcanza el valor de 1 y regresa dicho valor. Es entonces que la variable numero de la función de nivel inferior se multiplica por el parámetro numero de la función superior hasta llegar a la función de más alto nivel.

# Métodos

Un método *no es otra cosa que una función, la diferencia es que está definido dentro de cierta clase*.

Una función, como su nombre lo indica, cumple determinada tarea al ser ejecutada, un método, entonces, también.

Cada tipo de dato es una clase, por lo tanto, tienen sus propios métodos.

# Métodos de tipo INT

```
int()
abs()
divmod(a, b)
float()
mod(a, b)
neg()
pow(a, b)
round()
sizeof()
str()
bit_lenght()
sum(a, b)
bool()
```

# Métodos de tipo STRING

```
str()
“string”.__getitem__(i)
len()
capitalize()
“string”.__contains__(“a”)
isalnum()
isalpha()
lower()
upper()
title()
```

```
istitle()
“string”.join(*args)
strip()
“string”.replace(a, b)
“string”.rfind(a)
“string”.count(a)
“string”.find(a)
split()
splitlines()
```

# Métodos de tipo LIST

```
list()
lista.__add__([lista2])
lista.__contains__(a)
lista.__delitem__(i)
lista.__getitem__(i)
len()
lista.__setitem__(i, a)
lista.count(a)
```

```
lista.pop()
lista.sort()
list(lista.__reversed__())
sizeof()
lista.append(a)
lista.clear()
lista.copy()
lista.extend(a)
```

# Métodos de tipo DICT

```
dict()
diccionario.__delitem__(a)
diccionario.__contains__(a)
diccionario.__getitem__(a)
len(diccionario)
diccionario.__setitem__(a, b)
```

```
diccionario.items()
diccionario.keys()
diccionario.clear()
diccionario.copy()
diccionario.values()
sizeof(diccionario)
```

Para ver todos los métodos de todos los tipos de datos, pueden visitar la sección documentation de la web oficial de Python:
https://docs.python.org/3/

# Funciones predefinidas (built-in functions)

Las funciones predefinidas, en inglés built-in functions, son *aquellas que ya vienen incluidas en Python*. Por ejemplo, los métodos vistos anteriormente, no son otra cosa que funciones predefinidas.

Que ya estén incluidas en Python quiere decir que no se necesita incluir ningún módulo, paquete o librería para poder utilizarlas.

- all() → devuelve True si todos los elementos del objeto iterable que se le pasa por parámetro tienen un valor booleano True, False si tienen al menos un valor booleano False, 0 o vacío (que representa False).
- any() → devuelve True si al menos uno de los elementos del objeto iterable que se le pasa por parámetro tiene un valor booleano True.
- ascii() → devuelve el valor ASCII que corresponde al dato que se le pasa por parámetro.

- `bin()` → convierte en binario el entero que se le pasa por parámetro.
- `bool()` → convierte en booleano el elemento que se le pasa por parámetro.
- `bytearray()` → devuelve un array del número de bytes que se pasaron por parámetro.
- `byte()` → devuelve un objeto byte inmutable del elemento que se le pasó por parámetro.
- `callable()` → devuelve True si el objeto pasado por parámetro es “llamable”, las funciones son llamables, por ejemplo.

- `compile()` → convierte un string en un code object.
- `delattr()` → toma dos parámetros, una clase y un atributo de ella y elimina el atributo.
- `dir()` → devuelve todos los atributos del objeto que se le pase por parámetro.
- `enumerate()` → agrega un contador al objeto iterable.
- `eval()` → toma un string como parámetro y lo convierte en una expresión.
- `filter()` → se le pasan dos parámetros, una condición y un elemento iterable y devuelve aquellos en los cuales se cumple la condición.

- frozenset() → devuelve un conjunto inmutable del elemento que se le pasa por parámetro.
- getattr() → se le pasan dos parámetros, una clase y un atributo, y devuelve el valor de ese atributo.
- hasattr() → se le pasan dos parámetros, una clase y un atributo, y devuelve True si dicha clase tiene ese atributo.
- hash() → devuelve el valor hash del dato que se pasa por parámetro.
- hex() → convierte a hexadecimal el dato que se le pasa por parámetro.

- id() → devuelve el valor identidad del dato que se le pasa por parámetro.
- isinstance() → se pasan dos parámetros, un dato y un tipo de objeto y devuelve True si ese dato pertenece a ese objeto.
- issubclass() → se pasan dos parámetros (dos clases) y devuelve True si la primera es subclase de la segunda.
- iter() → devuelve un iterador Python.
- map() → se le pasan dos parámetros, una función y un elemento iterable, y devuelve True o False en cada iteración si se cumple o no la función respectivamente.

- `max()` → devuelve el valor máximo de un conjunto.
- `min()` → devuelve el valor mínimo de un conjunto.
- `next()` → devuelve el próximo elemento del objeto iterable que se pasa por parámetro.
- `oct()` → convierte a octal el dato que se le pasa por parámetro.
- `open()` → abre el archivo de la ruta de acceso que se pasa por parámetro.
- `type()` → devuelve el tipo al que pertenece el dato pasado por parámetro.
- `zip()` → devuelve un objeto iterable de tuplas.

- `ord()` → devuelve el valor Unicode al que pertenece el dato que se le pasó por parámetro.
- `range()` → crea una lista de elementos consecutivos, si se le pasa un sólo n parámetro es de 0 a n-1, si se le pasan dos parámetros n1 y n2, el rango es de n1 a n2-1, si se le pasan 3 parámetros n1, n2 y n3, el rango es de n1 a n2-1 y el salto es de n3 en n3.
- `set()` → devuelve un conjunto del dato que se le pasó por parámetro.
- `slice()` → devuelve los elementos del dato que forman parte de esa porción de dato, puede tener 2 o 3 parámetros, al igual que range.


# Comprensión de listas (list comprehension)

Hay diferentes formas de crear listas, pero, para entenderlas mejor existe la forma de list comprehension, es *una suerte de combinación de listas con funciones y/o bucles*.

Con este tipo de sintaxis se puede trabajar con cualquier tipo de dato, e incluso con archivos. Se utilizan mucho para filtrar data de archivos en data science.

```python
mi_lista = [pow(i, 2) for i in range(0, 21)]

print(mi_lista)

mi_string = "Hola 123456 chau"

mi_lista = [x for x in mi_string if x.isdigit()]
print(mi_lista)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400]
['1', '2', '3', '4', '5', '6']
```