

Unidad 3

# Funciones y Programación Modular

Manual del Estudiante

CLOUD  
FORMACIÓN

Maria del Carmen Sierra Fernández

## Contenido

Nivel I: Programación con Python (cubre Examen 98-381) .....	2
Unidad 3: Funciones y Programación Modular.....	2
Funciones.....	2
Definición de una función .....	2
docstrings .....	3
Cómo invocar a una función.....	4
Cómo organizar el código.....	5
Parámetros .....	7
Retornar un valor (return).....	9
Decodificación de una función .....	12
Funciones que llaman a otras funciones (Composición) .....	13
Funciones dentro de funciones .....	13
Funciones Recursivas.....	14
Alcance de las Variables .....	15
Variables globales.....	16
Parámetros opcionales.....	17
Parámetros claves (Keyword).....	19
Funciones con argumentos variables, *args y *kwargs en Python.....	19
Uso de *args ( args viene de arguments en Inglés).....	20
Uso de **kwargs .....	20
Funciones Anónimas o Lambda.....	22
Funciones que utilizan funciones lambda .....	23
Módulos Python .....	24
Sentencia import .....	26
Localizando módulos.....	28
Paquetes Python .....	28
Sentencia from .....	30
Bibliografía.....	32

## Nivel I: Programación con Python (cubre Examen 98-381)

### Unidad 3: Funciones y Programación Modular

#### Funciones

Una función la podemos definir como un fragmento de código que realiza alguna operación que es significativa. Las funciones en Python constituyen unidades lógicas de un programa y tienen un doble objetivo:

1. Dividir y organizar el código en partes más sencillas.
2. Encapsular el código que se repite a lo largo de un programa para ser reutilizado.

Python como hemos visto incluye un conjunto de funciones que podemos utilizar directamente en nuestras aplicaciones. Por ejemplo, la función `len()`, que obtiene la longitud o el número de elementos de un objeto como una lista, una tupla, un diccionario o un conjunto o la función `print()`, que muestra por consola un texto.

Ahora vamos a ver cómo definir nuestras propias funciones para estructurar el código de manera que sea más legible y reutilizar aquellas partes que se repiten a lo largo de una aplicación.

El siguiente video en inglés, explica las funciones de una manera muy amena y gráfica:  
<https://youtu.be/85WYZKKnaPQ>

#### Definición de una función

La sintaxis para crear una función, es:

```
def nombre(parámetros):  
    sentencias
```

El nombre de una función deben seguir las reglas vista anteriormente: comenzar con una letra, no puede comenzar por un número, no puede usar una palabra clave o reservada de Python, etc.

Los parámetros especifican qué información, si la hay, debemos proporcionar para utilizar la nueva función. Es decir, los parámetros especifican lo que **la función necesita para hacer su trabajo**.

Puede haber cualquier cantidad de sentencias o declaraciones dentro de la función, pero deben estar “indentadas” (la sangría estándar de cuatro espacios).

Las definiciones de funciones tienen el siguiente patrón:

1. Una línea de encabezado que comienza con la palabra clave y termina con dos puntos (:).

2. El cuerpo de la función, que consta de una o más sentencias o declaraciones Python, cada una con la misma sangría desde la línea del encabezado.

En la definición de una función, la palabra clave en el encabezado es `def`, que va seguida del nombre de la función y algunos nombres de `parámetros` entre paréntesis. La lista de `parámetros` puede estar vacía o puede contener cualquier número de `parámetros` separados entre sí por comas. En cualquier caso, se requieren los paréntesis.

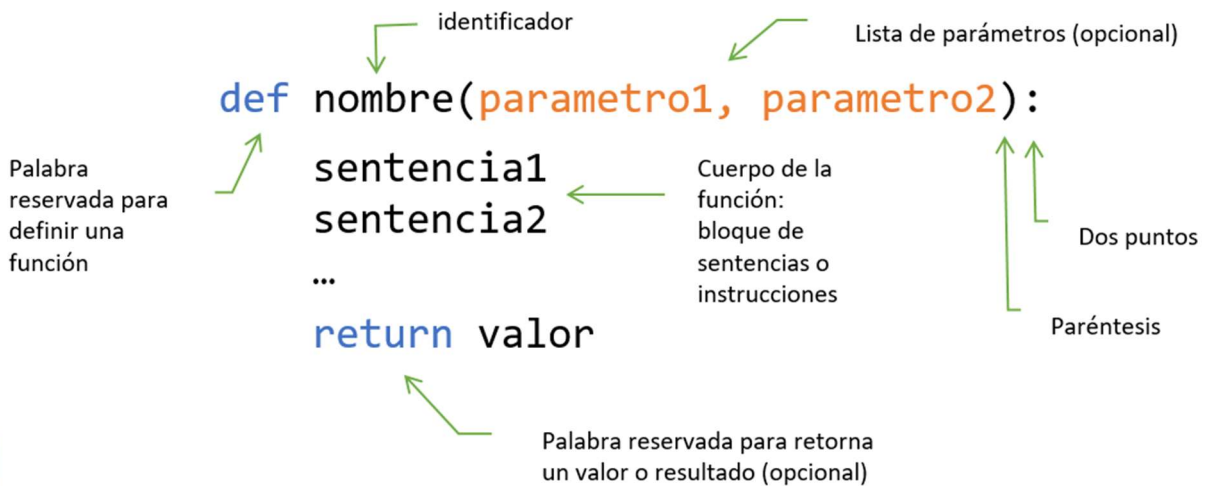


Figura 1: Sintaxis de una Función en Python

#### Ejemplo:

```
def saludo():
    """Esta función muestra un saludo"""
    print("Hola, bienvenidos al Curso de Python")
    print("...ahora vamos a ver funciones")

print(type(saludo))
<class 'function'>
```

#### docstrings

Si lo primero que aparece después del encabezado de la función, es una cadena encerrada entre tres comillas simples `'''` o dobles `"""`, se llama `docstring` y recibe un tratamiento especial en Python. El `docstring` es una cadena que se utiliza para documentar la función, es decir, indicar qué hace dicha función.

Otra forma de recuperar esta información es usar el intérprete interactivo e ingresar la expresión `<nombre_función> .__ doc__`, que recuperará el `docsstring` para la función. Por lo tanto, las herramientas de Python pueden recuperar la cadena que escribe como documentación al comienzo de una función en tiempo de ejecución. Esto es diferente de los comentarios en su código, que se eliminan por completo cuando se analiza el programa.

Por convención, los programadores de Python utilizan docstring para la documentación clave de sus funciones.

### Cómo invocar a una función

Definir una nueva función no hace que la función se ejecute. Para ejecutar la función, necesitamos llamar a la función. Esto también se conoce como invocar a la función.

La forma de invocar una función es referirse a ella por su nombre, seguido de paréntesis. Una vez que hemos definido una función, podemos llamarla tantas veces como queramos y sus declaraciones se ejecutarán cada vez que la llamemos.

### # Ejemplo Función Saludo

```
def saludo():  
    """Esta función muestra un saludo"""  
    print("Hola, bienvenidos al Curso de Python")  
    print("...ahora vamos a ver funciones")
```

```
saludo() # aquí estamos llamando a la función saludo()  
Hola, bienvenidos al Curso de Python  
...ahora vamos a ver funciones
```

### # Ejemplo Multiplica un número por 5

#### # Declaración de funciones

```
def multiplica_x_5(numero):  
    """Multiplica por 5 un número."""  
    print(f'La función ejecuta: {numero} * 5 = {numero * 5}')
```

#### # Programa principal

```
print('Comienzo del programa')  
print('-----')
```

#### # llamamos la función, pasando el argumento 125

```
multiplica_x_5(125)  
print('-----')  
print('Volvemos a llamar a la función')  
print('-----')
```

#### # llamamos la función, pasando el argumento 250

```
multiplica_x_5(250)  
Comienzo del programa  
-----  
La función ejecuta: 125 * 5 = 625  
-----
```

Volvemos a llamar a la función

-----

La función ejecuta:  $250 * 5 = 1250$

### *Cómo organizar el código*

Antes de continuar profundizando en las funciones, es importante conocer donde debemos ubicar las funciones en nuestro código. El código lo organizaremos en el siguiente orden:

1. Las librerías que vamos a utilizar: primero las librerías estándar, seguidas de las librerías de terceras partes y aplicaciones locales
2. Las funciones que utilizaremos en nuestro código
3. Nuestro código o programa principal, desde donde invocaremos nuestras funciones, etc.

Veamos un ejemplo:

#### # Librerías a utilizar

```
import turtle
```

#### # Funciones

```
def dibuja_cuadrado(t, pasos):
```

```
    """ La tortuga t dibujara un cuadrado
        cada con lado con un tamaño igual a pasos.
    """
```

```
    for i in range(4):
        t.forward(pasos)
        t.left(90) # ángulo del giro
```

#### # Código o Programa Principal

```
wn = turtle.Screen()      # define los atributos de la ventana
alex = turtle.Turtle()    # crea una tortuga llamada alex
```

```
# Llama a la función pasando los argumentos alex y 150
```

```
dibuja_cuadrado(alex, 150)
```

```
wn.exitonclick() # cierra la ventana al hacer clic
```

## IDE PyCharm

```
# Librerías a utilizar

import turtle

# Funciones

def dibuja_cuadrado(t, pasos):

    """ La tortuga t dibujará un cuadrado
        cada con lado con un tamaño igual a pasos.
    """
    for i in range(4):
        t.forward(pasos)
        t.left(90) # ángulo del giro

# Código o Programa Principal

wn = turtle.Screen() # define los atributos de la ventana
alex = turtle.Turtle() # crea una tortuga llamada alex
# Llama a la función pasando los argumentos alex y 150
dibuja_cuadrado(alex, 150)
wn.exitonclick() # cierra la ventana al hacer clic
```

## Jupyter Notebook

### Cómo organizar el código

```
In [1]: 1 # Librerías a utilizar
        2
        3 import turtle
        4
```

```
In [2]: 1 # Funciones
        2
        3
        4 def dibuja_cuadrado(t, pasos):
        5
        6     """ La tortuga t dibujara un cuadrado
        7         cada con lado con un tamaño igual a pasos.
        8     """
        9     for i in range(4):
        10         t.forward(pasos)
        11         t.left(90) # ángulo del giro
        12
        13 def funcion2 ()
        14     ....
        15
```

```
In [3]: 1 # Bloque del Código o Programa Principal
        2
        3 wn = turtle.Screen() # define los atributos de la ventana
        4 alex = turtle.Turtle() # crea una tortuga llamada alex
        5 # Llama a la función pasando los argumentos alex y 150
        6 dibuja_cuadrado(alex, 150)
        7 wn.exitonclick() # cierra la ventana al hacer clic
        8
```

```
In [ ]: 1 # Bloque del Código o Programa Principal
```

### Parámetros

Una función recibe unas entradas (inputs) y produce un resultado. Las entradas a una función se llaman parámetros y son conocidos. Los parámetros definidos en la función se conocen como **parámetros formales o nombres de parámetros**. Estos nombres pueden ser cualquier nombre de variable válido. Si hay más de uno, están separados por comas.

En la invocación de la función, entre paréntesis se debe proporcionar un valor para cada uno de los nombres de los parámetros, estos se conocen como **parámetros actuales o argumentos**. Estos valores están separados por comas. Los valores se pueden especificar directamente o mediante cualquier expresión de Python, incluida una referencia a algún otro nombre de variable.

Así es, como siempre funcionan las invocaciones de funciones. Cada una de las expresiones, separadas por comas, que están entre paréntesis se evalúa para producir valores. Luego, esos valores se comparan posicionalmente con los parámetros formales. El primer nombre de parámetro está vinculado al primer valor proporcionado. El nombre del segundo parámetro está vinculado al segundo valor proporcionado.

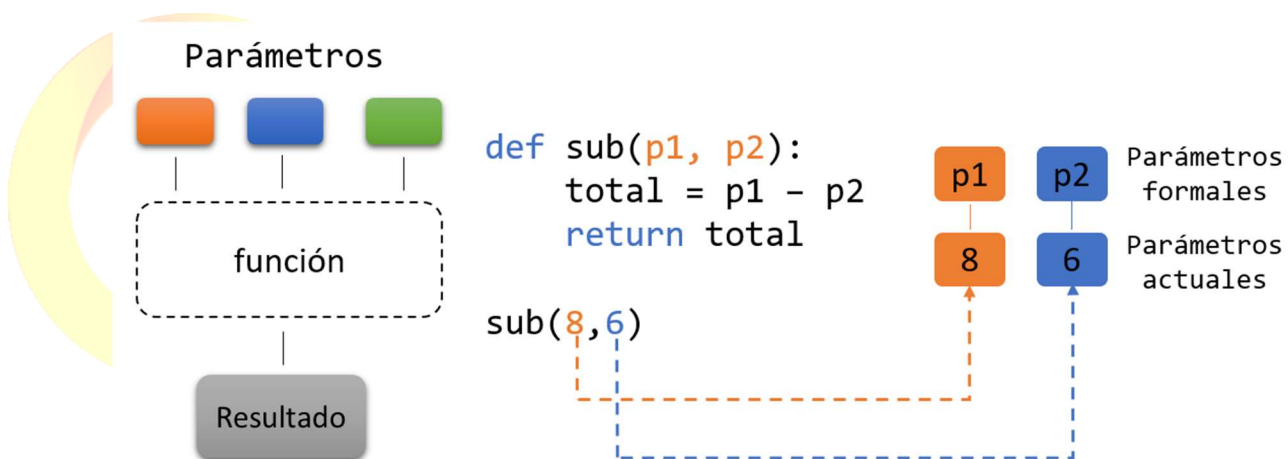


Figura 2: Asignación de los argumentos o parámetros actuales a los parámetros formales

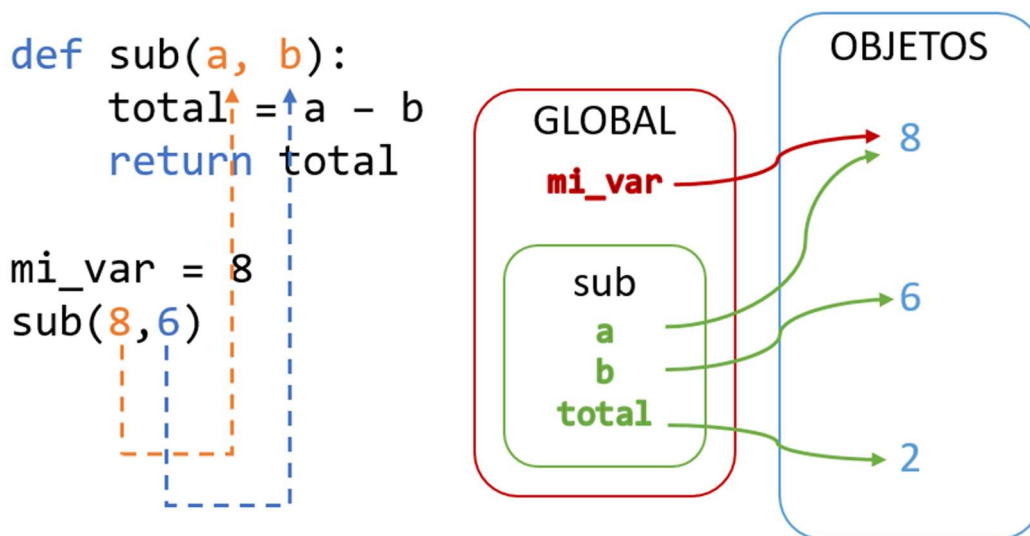


Figura 3: Variables y parámetros



Lo primero que se hace cuando invocamos una función, es evaluar los parámetros. En la figura 3 observamos que la variable `mi_var` tiene asignado el valor 8 y que los argumentos que pasamos a la función `sub()` son 8 y 6 respectivamente. Esto significa que la variable `a` apunta a 8 y `b` apunta a 6, cuando se ejecuta el código de la función `total = a - b`, entonces `total` apunta al valor calculado por la función, es decir 2.

### # Función restar 2 números

```
def sub(a, b):
    total = a - b
    return total

mi_var = 8
tresta = sub(8,6)
print(tresta)
2
```

¿Qué pasa si intentamos utilizar `mi_var` dentro de la función? La función `sub` puede obtener el valor asignado a `mi_var`, porque es una variable global (aunque no es recomendable utilizarlas de esta forma, la mejor práctica es pasarla como un parámetro)

### # Función restar 2 números

```
def sub(a, b):
    total = a - b
    print(mi_var) # mostrar una variable global
    return total

mi_var = 8
tresta = sub(mi_var,6)
print(total) # mostrar una variable local de la función
print(tresta)
```

¿Sucedre lo mismo con la variable `total`? No, las variables creadas dentro de la función son variables locales de la función y no podemos utilizarlas fuera de la misma. En el ejemplo anterior, el programa genera un error, porque la variable `total` no está definida:

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-7-26b691a49a57> in <module>
      8 mi_var = 8
      9 tresta = sub(mi_var,6)
----> 10 print(total)
      11 print(tresta)

NameError: name 'total' is not defined
```

El siguiente video, explica como las funciones toman los valores, los asignan a los parámetros y nos ayuda a comprender como funcionan las variables de la función y las variables de nuestro programa. [https://youtu.be/Ndw\\_EgFO\\_tw](https://youtu.be/Ndw_EgFO_tw)

### Retornar un valor (return)

A una función, no solo podemos pasar un valor de parámetro, una función también puede producir y retornar un valor.

Las funciones que devuelven valores a veces se denominan en inglés “fruitful functions” o funciones fructíferas en español. En muchos otros lenguajes, una función que no devuelve un valor se llama procedimiento, en Python se denominan funciones, independientemente si retornan o no un valor.

La declaración `return` va seguida de una expresión que se evalúa. Su resultado se devuelve a quién realizó la invocación de la función. Es importante resaltar, que todas las funciones Python devuelven el valor especial `None`, a menos que haya una declaración de retorno explícita con un valor distinto de `None`.

#### Ejemplo de una función con return:

```
# Funciones

def calcular_cuadrado(x):
    y = x * x
    return y # Esta función retorna un valor

# Programa

numero = 10
resultado = calcular_cuadrado(numero)
print("El cuadrado de {} es {}".format(numero, resultado))
El cuadrado de 10 es 100.
```

#### Ejemplo de una función ( return None):

```
# Funciones

def calcular_cuadrado(x):
    y = x * x
    print("El resultado es: ", y) # Muestra el valor

# Programa

numero = 10
resultado = calcular_cuadrado(numero)
print("El cuadrado de {} es {}".format(numero, resultado))
El resultado es: 100
El cuadrado de 10 es None.
```

¿Puede una función retornar más de un valor? La sentencia `return` sólo puede retornar un único valor u objeto. Si necesitamos retornar más de un valor podemos utilizar el objeto lista.

```
import random

def generar_numeros_aleatorios():
    """
    La función que genera 100 números pseudo aleatorios entre 0
    y 1000.
    Retorna una lista con los 100 números generados
    """
    lista = []
    for i in range(100):
        lista.append(random.randint(0, 1000))
    return lista

numeros = []
numeros = generar_numeros_aleatorios()
print(numeros)
```

```
[215, 643, 24, 185, 450, 32, 315, 787, 19, 792, 768, 361, 494, 138,
873, 988, 587, 672, 918, 349, 395, 612, 52, 278, 945, 126, 943, 999,
991, 406, 139, 302, 495, 467, 810, 337, 175, 473, 137, 857, 733,
307, 512, 144, 740, 448, 67, 295, 197, 325, 781, 741, 374, 58, 735,
752, 762, 825, 118, 740, 370, 454, 259, 259, 677, 269, 226, 227,
581, 974, 573, 856, 356, 617, 190, 675, 530, 450, 476, 77, 538, 608,
630, 43, 741, 918, 866, 483, 86, 746, 290, 213, 501, 983, 214, 543,
470, 482, 570, 787]
```

El siguiente video, explica de forma muy simple, como una función retorna un valor  
<https://youtu.be/LGOZyrRCJ1o>.

Veamos ahora un ejemplo con varias funciones:

#### # Librerias

```
import math
```

#### # Funciones

```
def area_cuadrado(lado):
    a_cuadrado = lado**2
    return a_cuadrado

def area_triangulo(base, altura):
    a_triangulo = (base*altura)/2
    return a_triangulo

def area_circulo(radio):
    a_circulo = math.pi*radio**2
    return a_circulo
```

```
# Programa
```

```
# Calculemos el área de un cuadrado invocando a la función  
area_cuadrado
```

```
l = 25  
area = area_cuadrado(l)  
print(f"El área de un cuadrado de {l} es {area:.2f}.")
```

```
# Calculemos el área de un triángulo invocando a la función  
area_triangulo
```

```
b = 25  
h = 20  
area = area_triangulo(b,h)  
print(f"El área de un triángulo con base {b} y altura {h} es  
{area:.2f}")
```

```
# Calculemos el área de un triángulo invocando a la función  
area_triangulo
```

```
b = 3  
h = 7  
area = area_triangulo(b,h)  
print(f"El área de un triángulo con base {b} y altura {h} es  
{area:.2f}")
```

```
# Calculemos el área de un círculo invocando a la función  
area_circulo
```

```
r = 13  
area = area_circulo(r)  
print(f"El área de un círculo con radio {r} es {area:.2f}")
```

Ahora es tu turno!

### Ejercicio:

1. Copia el programa anterior, y crear las funciones para calcular el área de las figuras: rectángulo, trapecio, rombo, paralelogramo y polígono regular
2. Invoca las funciones necesarias para calcular el área de la siguiente figura geométrica compuesta (Ver figura 4):

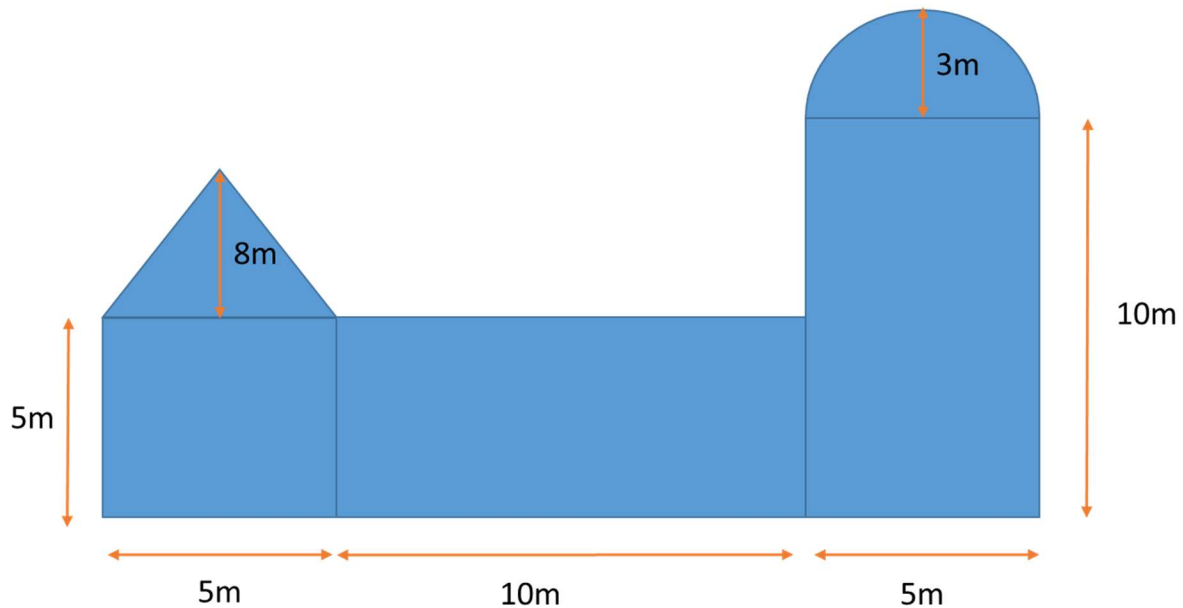


Figura 4: Ejercicios cálculo del área de una figura compuesta

### Decodificación de una función

Para desarrollar la comprensión de cualquier función, debemos intentar responder las siguientes preguntas:

1. ¿Cuántos parámetros tiene?
2. ¿Cuál es el tipo de valores que se pasarán cuando se invoca la función?
3. ¿Cuál es el tipo de valor de retorno que produce la función cuando se ejecuta?

La primera pregunta siempre es fácil de responder. Mire la línea con la definición de la función, mire dentro del paréntesis y cuente cuántos nombres de variable hay. La segunda y tercera preguntas no siempre son tan fáciles de responder. En Python, a diferencia de otros lenguajes de programación, no se declara que las variables tengan tipos fijos, y lo mismo ocurre con los nombres de las variables que aparecen como parámetros formales de funciones. Tienes que descifrarlo desde el contexto. Para averiguar los tipos de valores que una función espera recibir como parámetros, puede ver las invocaciones de funciones o puede ver las operaciones que se realizan en los parámetros dentro de la función.

Por esta razón es importante documentar nuestras funciones, respondiendo también a estas preguntas.

Ejemplo:

```
def es_par(n):
    """
    La función es_par(n) recibe como parámetro n que es un
    número entero, y retorna un valor booleano.
    Si n es un numero par retorna True y
    Si n no es par o no es un número entero retorna False.
    """
```

```
if n%2 == 0:
    return True
else:
    return False

print(es_par(12))
print(es_par(15))
True
False
```

### Funciones que llaman a otras funciones (Composición)

Cada una de las funciones que escribimos se puede usar e invocar desde otras funciones. Este proceso nos permite dividir un problema en problemas más pequeños. Este proceso se conoce como descomposición funcional.

```
def calcular_cuadrado(x):
    y = x * x
    return y

def sumar_cuadrados(x,y,z):
    c1 = calcular_cuadrado(x)
    c2 = calcular_cuadrado(y)
    c3 = calcular_cuadrado(z)
    return c1 + c2 + c3

# LLama a la función suma de los cuadrados

a = -5
b = 2
c = 10

resultado = sumar_cuadrados(a,b,c)
print(resultado)
129
```

En el ejemplo observamos, como la función `sumar_cuadrados` llama a la función `calcular_cuadrado` y el valor que retorna la función lo asigna a la variable `c1`, `c2` y `c3` y esta a su vez retorna la suma a la variable `resultado`.

### Funciones dentro de funciones

Python permite crear una función dentro de otra función, pero la vida de esas funciones es efímera porque sólo pertenece al ámbito donde fue definida, es decir, que será creada cada vez que llamemos a la función raíz y luego dejará de existir. Además esto implica que no podemos llamarla desde fuera de la función que la crea. Son pocos los casos en los que esto es útil, principalmente en clausuras y factory-functions.

```
def extraer_usuario_dominio(email):
    import re
    resultado = []
    def extraer_usuario():
        patron = '([^\s]*)@'
        resultado.append(re.findall(patron, email))
    def extraer_dominio():
        patron = '@([^\s]*)'
        resultado.append(re.findall(patron, email))

    extraer_usuario()
    extraer_dominio()
    return resultado

correo = "m.sierra@cloudftic.com"
print(extraer_usuario_dominio(correo))
```

### Funciones Recursivas

La recursividad en programación es una técnica muy potente, ésta se realiza con funciones que se llaman a sí mismas, cómo una especie de bucle, porque el mismo código se repetirá varias veces, hasta llegar a la solución.

#### Ejemplo Factorial de un número:

Podemos definir el factorial de un número entero positivo  $n$ , expresado  $n!$ , como el producto de todos los números enteros positivos menores o iguales que  $n$ . (Wikipedia, 2020)

$$n! = 1 \times 2 \times 3 \times 4 \times \dots \times (n-1) \times n.$$

También es posible definirlo mediante la relación de recurrencia

$$n! = \begin{cases} 1 & \text{si, } n = 0 \\ (n-1)! \times n & \text{si, } n > 0 \end{cases}$$

La segunda definición incorpora la premisa de que:  $0! = 1$  y  $1! = 1$ .

```
def factorial(numero):
    if(numero == 0 or numero == 1):
        return 1
    else:
        return numero * factorial(numero-1)

if __name__ == "__main__":
    try:
        print("Calculo de un número Factorial")
        num = int(input("Introduce un número positivo: "))
        if(num < 0):
            print("El número debe ser mayor o igual a 0")
        else:
```

```

        print(f"El factorial de {num} o {num}! es {factorial(num)}")
except:
    print("Se espera un número")

```

### Calculo de un número Factorial

Introduce un número positivo: 5

El factorial de 5 o 5! es 120

### Alcance de las Variables

El “Universo” de Python está dividido en un espacio para los objetos (object space) y otro espacio para los nombres de objetos o namespaces. El namespaces tiene tres partes lógicas:

1. **Built-in:** contiene los nombres de todas las funciones y variables que son definidas cuando se ejecuta Python. Por ejemplo: `print()`, `len()`, `id()`
2. **Global:** contiene los nombres y funciones que son declaradas en el programa principal.
3. **Local:** contiene las variables y parámetros que son declarados localmente, Por ejemplo: En una función.

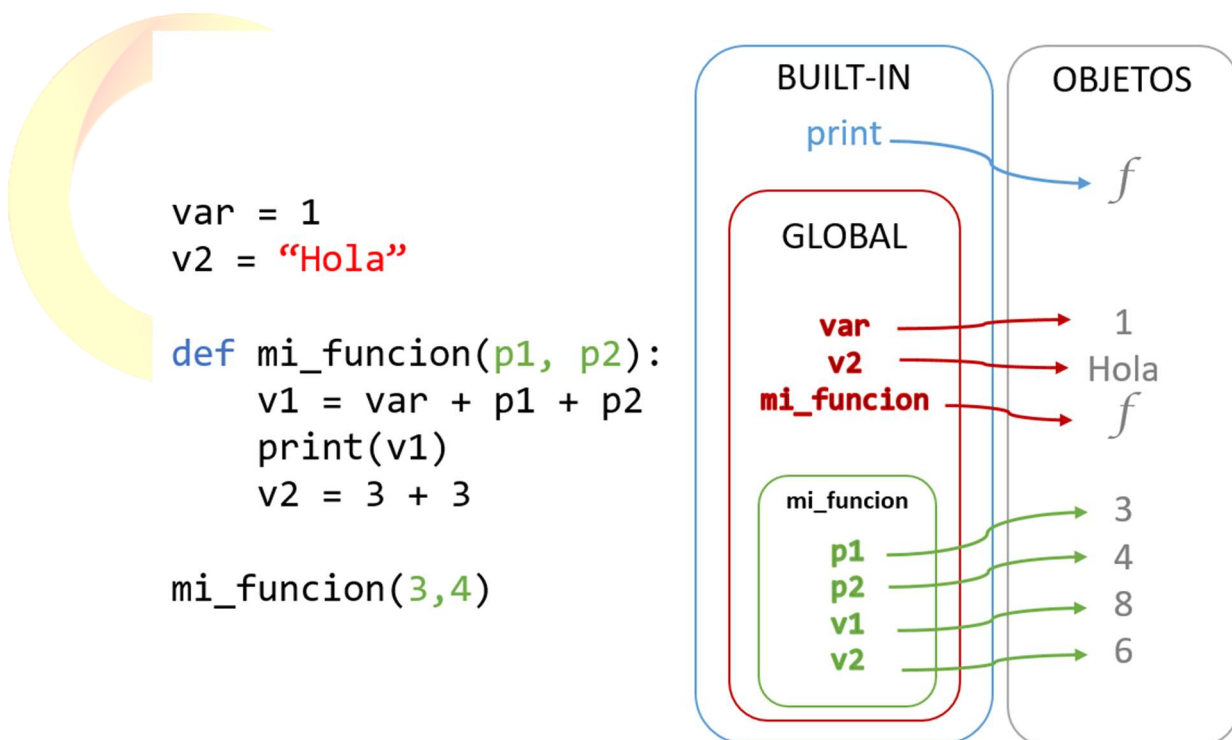


Figura 5: Alcance de las variables

Una instrucción de asignación en una función, crea una variable local para la variable al lado izquierdo del operador de asignación. Se llama local, porque esta variable solo existe dentro de la función y no puede utilizarse fuera de la función.

Las variables locales, solo existen mientras se ejecuta la función; a esto lo llamamos su vida útil. Cuando la ejecución de la función termina (retorna un valor), las variables locales se destruyen. Los parámetros formales también son locales y actúan como variables locales.



En nuestro ejemplo las variables locales son: v1, v2, p1 y p2. Observe, que la variable local v2 tiene un valor de 6, mientras que la variable v2 global tiene asignado el valor “Hola”.

### Variables globales

Los nombres de variables que están en el nivel superior, no dentro de ninguna definición de función, se denominan globales.

Aun cuando una función puede acceder a una variable global, esto se considera de mala práctica y debe evitarse.

En nuestro ejemplo de la figura 5, observamos las variables globales son: mi\_función, var y v1 con el valor de “Hola”. Observe que tenemos dos variables v1 (una global y otra local) para Python son dos variables diferentes.

Ahora veamos, cómo Python busca las variables:

Primero, Python mira las variables que están definidas como variables locales en la función. A esto lo llamamos ámbito local. Si el nombre de la variable no se encuentra en el ámbito local, entonces Python mira las variables globales o el ámbito global.

Hay otra variación en este tema de variables locales versus globales. Las declaraciones de asignación en la función local no pueden cambiar las variables definidas fuera de la función.

Cuando una variable local tiene el mismo nombre que una variable global, decimos que lo local oculta lo global. Esto significa, que Python no puede acceder a la variable global porque la variable local se encontrará primero. Esta es otra buena razón para no utilizar variables globales.

Si realmente desea cambiar el valor de una variable global dentro de una función, puede hacerlo declarando explícitamente que la variable es global. Esto no debe hacerse, es un ejemplo para la comprensión de cómo funciona Python.

```
def mi_funcion(x,p):  
    global var1
```

Recuerde, generalmente, no es recomendable hacer variables globales. El siguiente video explica con detalle cómo se realiza la asignación de los valores a cada variable y en cada espacio. <https://youtu.be/HdFujZpLFVg>

Resumiendo, cuando trabajamos con funciones, es muy importante saber el orden en el que se ejecutan las sentencias o el flujo de ejecución.

La ejecución siempre comienza en la primera declaración del programa. Las declaraciones se ejecutan una a la vez, en orden, de arriba a abajo. Las definiciones de funciones no alteran el flujo de ejecución del programa, pero recuerde que las declaraciones dentro de la función no se ejecutan hasta que se llama a la función. Las llamadas a funciones son como un desvío en

el flujo de ejecución. En lugar de ir a la siguiente declaración, el flujo salta a la primera línea de la función llamada, ejecuta todas las declaraciones allí y luego regresa para continuar donde lo dejó. El intérprete es un experto en realizar seguimiento de dónde se encuentra inclusive cuando una función llama a otra función, el programa retoma donde lo dejó en la función que la llamó. Cuando llega al final del programa, termina.

### Parámetros opcionales

En el tratamiento de las funciones hasta ahora, cada definición de función específica cero o más parámetros formales y cada invocación de función proporcionan exactamente esa cantidad de valores. A veces es conveniente tener parámetros opcionales que se pueden especificar u omitir.

1. Cuando se omite un parámetro opcional al invocar una función, el parámetro formal se le asigna un valor predeterminado.
2. Cuando se incluye el parámetro opcional, entonces el parámetro formal se le asigna el valor proporcionado.

Los parámetros opcionales son convenientes cuando una función casi siempre se usa de una manera simple, pero es bueno permitir que se use de una manera más compleja, con valores no predeterminados especificados para los parámetros opcionales. Veamos un ejemplo con la función `int()`.

La sintaxis de la función `int()` es:

`int(x, base=10)`

donde:

- `x`: es un número o cadena que se convertirá en un objeto entero.
- `Base`: base del número en `x`. Por defecto la base es 10, pero puede tomar valores desde 2-36.

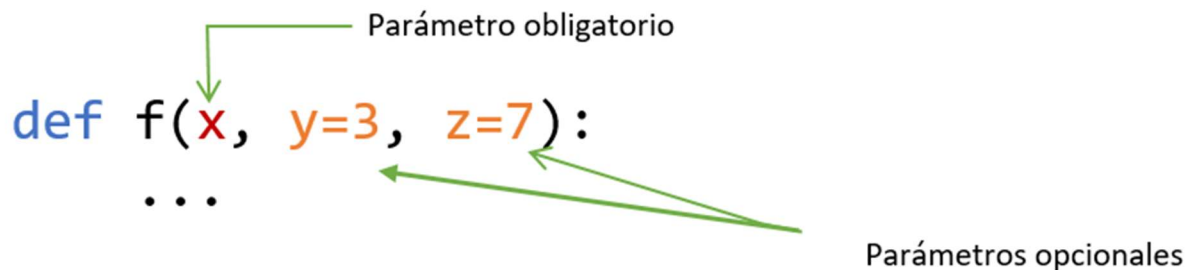
Ejemplo:

```
print(int("100"))  
print(int("100", 10))    # El resultado es el mismo, porque por  
defecto la base es 10  
print(int("100", 8))     # Si la base es 8, nos representa 100 en  
base 8  
print(int("100", 2))     # Si la base es 2, nos representa 100 en  
base 2  
print(int("100", 16))    # Si la base es 16, nos representa 100 en  
base 16  
100  
100  
64  
4
```

256

Al definir nuestra función, podemos especificar un valor predeterminado para un parámetro. Ese parámetro se convierte en un parámetro opcional cuando se llama a la función. La forma de especificar un valor predeterminado es con una declaración de asignación dentro de la lista de parámetros.

**Ejemplo:**



```
def f(x, y=3, z=7):
    ...
```

Figura 6: Parámetros opcionales

**# Funciones**

```
def f(x, y=3, z=7):
    print("x, y, z, son: " + str(x) + ", " + str(y) + ", " + str(z))
```

**# Programa principal**

```
f(2)           # x es el parámetro obligatorio x = 2, y = 3 y z = 7
f(3, 5)        # x es el parámetro obligatorio x = 3, y = 5 y z = 7
f(2, 5, 8)     # x es el parámetro obligatorio x = 2, y = 5 y z = 8
```

Observamos, que el parámetro `x` siempre debe suministrarse porque es el parámetro obligatorio y por ende no tiene un valor por defecto, en el caso de los parámetros opcionales `y`, `z` observamos que cuando no se incluyen sus argumentos al invocar la función, estos toman los valores por defecto, pero cuando se especifican toman los valores de los argumentos.

**# Ejemplo**

```
def multiplicar_cadena(cadena, num = 3):
    return cadena*num

print(multiplicar_cadena("Hola "))
print(multiplicar_cadena("Es opcional, ",5))
```

Hola Hola Hola

Es opcional, Es opcional, Es opcional, Es opcional, Es  
opcional,

### Parámetros claves (Keyword)

Otra forma de invocar funciones con parámetros opcionales, es el paso de parámetros basados en palabras clave. Esto es particularmente conveniente cuando hay varios parámetros opcionales y desea proporcionar un valor para uno de los últimos parámetros sin proporcionar un valor para los anteriores.

Para pasar argumentos por palabra clave es muy simple. Al invocar una función, dentro del paréntesis siempre hay cero o más valores, separados por comas. Con argumentos de palabras clave, algunos de los valores pueden tener el formato:

```
nombredelparametro = valor
```

### Ejemplo:

#### # Funciones

```
def f(x, y=3, z=7):
    print("x, y, z, son: " + str(x) + ", " + str(y) + ", " + str(z))
```

#### # Programa

```
f(2)           # 1 argumento posicional
f(3, 5)        # 2 argumentos posicionales
f(2, 5, 8)      # 3 argumentos posicionales
f(2, z = 8)     # 1 argumento posicional, 1 argumento palabra clave
f(x = 2, z = 8) # 2 argumentos por palabra clave
f(x = 5, y=10, z = 17) # 2 argumentos por palabra clave
```

```
x, y, z, son: 2, 3, 7
x, y, z, son: 3, 5, 7
x, y, z, son: 2, 5, 8
x, y, z, son: 2, 3, 8
x, y, z, son: 2, 3, 8
x, y, z, son: 5, 10, 17
```

El siguiente código genera error, porque estamos pasando dos veces x por posición y por palabra clave:

```
f(2, x=10)
TypeError: f() got multiple values for argument 'x'
```

### Funciones con argumentos variables, \*args y \*kwargs en Python

Las funciones con argumentos variables, nos permiten definir funciones en las cuales no sabemos a priori cuantos parámetros vamos a recibir. Por ejemplo sumar un conjunto de datos.

### Uso de \*args ( args viene de arguments en Inglés)

Con \*args podemos definir funciones genéricas que no aceptan un número determinado de parámetros, sino que se "adaptan" al número de argumentos con los que son llamados.

\*args es una tupla, al cual podemos acceder iterando.

Ejemplo:

```
def suma(*args):  
    s = 0  
    for arg in args:  
        s += arg  
    return s  
  
suma(17, 993, 564, 584)  
2158
```

Una forma más simple

```
def suma(*nums): # podemos utilizar otro nombre  
    return sum(nums)  
  
suma(17, 993, 564, 584, 678, 908, 123)  
3867
```

### Uso de \*\*kwargs

\*\*kwargs en este caso tenemos un diccionario. Donde podemos dar un nombre a cada argumento: {nombre="Maria", apellido="Sanchez"}

Ejemplo:

```
def suma(**kwargs):  
    s = 0  
    for key, value in kwargs.items():  
        print(key, "=", value)  
        s += value  
    return s  
  
suma(a=56, b=88, c=94)  
a = 56  
b = 88  
c = 94  
238
```

Es posible mezclar argumentos normales con \*args y \*\*kwargs dentro de la misma función. Estos argumentos deben definirse en la función en el siguiente orden:

1. Primero argumentos normales.
2. Después los \*args.

3. Y por último los **\*\*kwargs**.

Ejemplo:

```
def funcion(a, b, *args, **kwargs):
    print("a =", a)
    print("b =", b)
    for arg in args:
        print("args =", arg)
    for key, value in kwargs.items():
        print(key, "=", value)

funcion(10, 20, 1, 2, 3, 4, x="Hola", y="que", z="tal")
a = 10
b = 20
args = 1
args = 2
args = 3
args = 4
x = Hola
y = que
z = tal
```

Haciendo uso de \*, podemos extraer los valores de una lista o tupla, y que sean pasados como argumentos a la función. Es lo que se conoce como tuple unpacking o desempaquetado de argumentos

```
def funcion(a, b, *args, **kwargs):
    print("a =", a)
    print("b =", b)
    for arg in args:
        print("args =", arg)
    for key, value in kwargs.items():
        print(key, "=", value)

nums = [1, 2, 3, 4]
cadenas = {'x':"Hola", 'y':"que", 'z':"tal"}

funcion(10, 20, *nums, **cadenas)

a = 10
b = 20
args = 1
args = 2
args = 3
args = 4
x = Hola
y = que
z = tal
```

### Funciones Anónimas o Lambda

En Python, las funciones lambda son también conocidas como funciones anónimas porque se definen sin un nombre. Una expresión lambda es una notación alternativa para crear una función. Python permite utilizar una sintaxis abreviada para declarar una función, se utiliza para aquellas funciones que consten de una única instrucción.

La sintaxis de una expresión lambda es:

**lambda** argumentos o parámetros: expresión

La palabra "lambda" seguida de los nombres de los parámetros, separados por comas pero no dentro (paréntesis), seguida de dos puntos y luego una expresión. La expresión produce un objeto de función, que se comporta como una función de una sola instrucción:

```
def nombre_funcion(parámetros):
    return valor_retorno
```

En la siguiente figura 7, observamos la equivalencia una función y una función lambda:

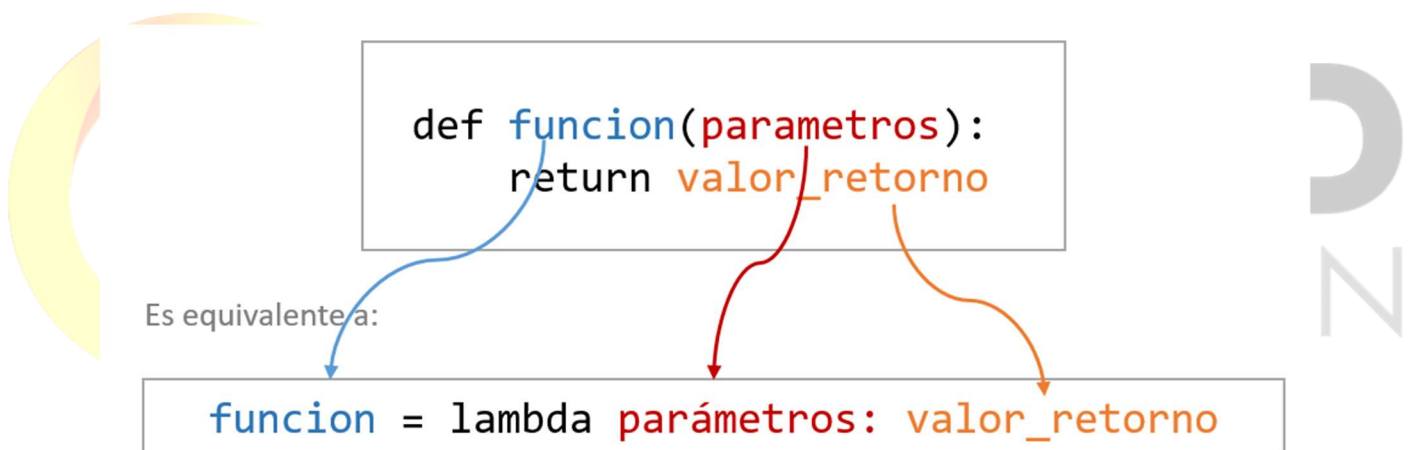


Figura 7: Funciones anónimas o lambda

La siguiente función extrae el último carácter de una cadena o string:

```
def extraer_ultimo_caracter(cadena):
    return cadena[-1]

print(extraer_ultimo_caracter("Piensa, cree, sueña y
atrévete"))
e
```

Ahora, vamos a crear su equivalente lambda:

```
extraer_ultimo_caracter = (lambda cadena: cadena[-1])

print(extraer_ultimo_caracter("Piensa, cree, sueña y
atrévete"))
```

Ambas funciones hacen exactamente lo mismo. En la función lambda `extraer_ultimo_caracter`, es un objeto función:

```
<function __main__.<lambda>(cadena)>
```

La siguiente función cambia el signo de un número

```
resultado = (lambda x: -x)

print(resultado(5))
-5

print(resultado(-5))
5
```

En las funciones lambda, también podemos utilizar parámetros opcionales y parámetros `*args` y `**kwargs`:

# Función lambda con parámetros opcionales

```
f = lambda x, y, z=1: (x + y) * z
f(5,6)
11
f(5,6,8)
88
```

# Función lambda con parámetros `*args`

```
suma = lambda *nums: sum(nums)

suma(17, 993, 564, 584, 678, 908, 123)
3867
```

**Funciones que utilizan funciones lambda**

`map()`: aplica una función a cada uno de los elementos de una lista.

```
enteros = [1, 2, 4, 7]
cuadrados = list(map(lambda x : x ** 2, enteros))
print(cuadrados)
[1, 4, 16, 49]
```

`filter()`: filtra una lista de elementos para los que una función devuelve True.

```
valores = [1, 2, 3, 4, 5, 6, 7, 8, 9]
impares = list(filter(lambda x : x % 2 != 0, valores))
print(impares)
[1, 3, 5, 7, 9]
```



`reduce()`: Esta función se utiliza principalmente para llevar a cabo un cálculo acumulativo sobre una lista de valores y devolver el resultado. La función `reduce()` está incluida en el módulo `functools`.

```
from functools import reduce

valores = [2, 4, 6, 5, 4]

suma = reduce(lambda x, y: x + y, valores)
print(suma)
```

21

Las funciones `sort()` y `sorted()` también se pueden utilizar con funciones `lambda`, las veremos con detalle en la Unidad 6 con las listas, diccionarios y tuplas.

### Módulos Python

Un módulo nos permite organizar lógicamente nuestro código Python. Agrupando código relacionado dentro de un módulo hace el código más fácil de entender y usar. Un módulo puede definir funciones, clases y variables, también puede incluir código ejecutable.

Un módulo es un fichero conteniendo definiciones y declaraciones de Python. Las definiciones de un módulo pueden ser importadas a otros módulos o al módulo principal. El nombre de archivo es el nombre del módulo con el sufijo `.py`. Dentro de un módulo, el nombre del mismo módulo (como cadena) está disponible en el valor de la variable global `__name__`.

Ahora, vamos a crear un módulo para el cálculo del área de diferentes figuras geométricas (cuadrado, triángulo, rectángulo, trapecio, paralelogramo, polígono regular y rombo), utilizando el código creado en el tema de funciones.

```
"""
Módulo para el cálculo del área de diferentes
figuras geométricas
"""

# Librerías

import math

# Funciones para cálculo del área de figuras geométricas

def area_cuadrado(lado):
    a_cuadrado = lado**2
    return a_cuadrado

def area_triangulo(base, altura):
    a_triangulo = (base*altura)/2
```

```
    return a_triangulo

def area_circulo(radio):
    a_circulo = math.pi*radio**2
    return a_circulo

def area_rectangulo(base, altura):
    a_rectangulo = base * altura
    return a_rectangulo

def area_trapecio(Base, altura, base):
    a_trapecio = ((Base + base)/2)*altura
    return a_trapecio

def area_paralelogramo(base, altura):
    a_triangulo = (base*altura)/2
    return a_triangulo

def area_poligono_regular(n_lados,lado, apotema):
    a_poligono = ((n_lados*lado)*apotema)/2
    return a_poligono

def area_rombo(DiagonalM, diagonalm):
    a_rombo = (DiagonalM*diagonalm)/2
    return a_rombo

if __name__ == "__main__":
    area = area_cuadrado(120)
    print(f"El área de un cuadrado con lados = 120 es: {area}")
```

**Pasos:**

1. Utilizando el IDE PyCharm, cree un nuevo archivo Python `calcular_areas_fg.py`, copie el código suministrado en el paso anterior y seleccione guardar.
2. En la parte inferior de la ventana de Pycharm seleccione Terminal. En el prompt escriba Python, debe aparecer el prompt `>>>`
3. Ahora vamos a verificar el directorio actual, desde el prompt `>>>` escriba:

```
>>> import os
>>> os.getcwd() # retorna el directorio actual
```

4. Verifique que el directorio actual sea el mismo donde creo el modulo `calcular_areas_fg.py`, si no es el mismo, cambie el directorio actual

```
>>> os.chdir(<escriba el directorio donde está su módulo .py>)
```

5. Importamos el módulo `calcular_areas_fg`

```
>>> import calcular_areas_fg
```

6. Ejecutemos el módulo para calcular el área de un círculo

```
>>> calcular_areas_fg.area_circulo(25)
1963.4954084936207
```

7. Ejecutemos el módulo para calcular otras áreas

```
>>> calcular_areas_fg.area_rectangulo(125, 250)
31250
```

```
>>> calcular_areas_fg.area_trapecio(20, 50, 40)
1500.0
```

8. Creemos una variable de tipo módulo `calcular_areas_fg` y realicemos algunas pruebas

```
>>> calcular = calcular_areas_fg
>>> calcular.area_poligono_regular(6,25,12)
900
>>> calcular.area_triangulo(25,15)
187.5
```

### *Sentencia import*

La sentencia `import` se utiliza para importar un módulo. Podemos invocar cualquier archivo de código Python como un módulo ejecutando esta sentencia en otro archivo de código Python. La sentencia `import` tiene la siguiente sintaxis:

```
import <nombre del módulo>
```

Cuando el intérprete encuentra una sentencia `import`, este importa el módulo si el mismo está presente en la ruta de búsqueda. Una ruta de búsqueda es una lista de directorios que el intérprete busca antes de importar un módulo.

Ahora, invocaremos nuestro modulo desde un programa Python, para importar el módulo `calcular_areas_fg.py`, necesitamos colocar la siguiente sentencia al inicio del otro script Python.

```
import calcular_areas_fg
```

*# Ejemplo Programa que importa el módulo `calcular_areas_fg`:*

```
import calcular_areas_fg
```

```
print("Cálculo del área de un cuadrado: {0} ".format(
    calcular_areas_fg.area_cuadrado(int(input("Ingrese
    el tamaño del lado de cuadrado: "))))
```

```
print("Cálculo del área de un círculo: {0} ".format(
    calcular_areas_fg.area_círculo(int(input("Ingrese
    el radio del círculo: "))))))
```

Ingrese el tamaño del lado de cuadrado: 25

Cálculo del área de un cuadrado: 625

Ingrese el radio del círculo: 50

Cálculo del área de un círculo: 7853.981633974483

Un módulo se carga solo una vez, independientemente de la cantidad de veces que se importe. Esto evita que la ejecución del módulo ocurra una y otra vez si se producen múltiples importaciones.

La primera vez que un módulo es importado en un script de Python, se ejecuta su código una vez. Si otro módulo importa el mismo módulo este no se cargará nuevamente; los módulos son inicializados una sola vez.

Para acelerar la carga de módulos, Python cachea las versiones compiladas de cada módulo en el directorio `__pycache__` bajo el nombre `module.version.pyc`, dónde la versión codifica el formato del archivo compilado; generalmente contiene el número de versión de Python.

*Variable `__name__`.*

Al final de nuestro modulo, hemos incluido la siguiente línea:

```
if __name__ == '__main__':
```

Pero ¿qué hace esta línea de código?

Cada vez que el intérprete de Python lee el código fuente de un módulo `.py`, ejecuta todo el código dentro de ese archivo, incluso, cuando lo importamos.

Sin embargo, en ocasiones no queremos ejecutar todo el código de un módulo, simplemente queremos importar alguna clase o función. Para evitar que el intérprete de Python ejecute todo el código del programa, utilizamos la variable `__name__`.

Esta variable suele contener los siguientes valores:

- `__main__`: cuando nuestro modulo es ejecutado como programa principal, el intérprete asigna el valor `__main__` a la variable `__name__`. Y ejecutará todo el código del módulo.
- Nombre del archivo: cuando importamos un módulo, el intérprete busca el archivo `.py` y antes de ejecutar el módulo, asigna el nombre del archivo a la variable `__name__`.

En resumen, podemos utilizar `if __name__ == '__main__':` para proteger parte del

código. De esta forma, podemos estar seguros de que ese bloque, solo se ejecutará cuando el módulo es el programa principal.

### *Localizando módulos*

Cuando importamos un módulo, el intérprete Python busca por el módulo en la secuencia siguiente:

1. El directorio actual.
2. Si el módulo no es encontrado, Python entonces busca en cada directorio en la variable de entorno `PYTHONPATH` del sistema operativo.
3. Si todas las anteriores fallan, Python busca la ruta predeterminada.

La ruta de búsqueda de módulo es almacenada en el módulo `sys`, como la variable `sys.path`. La variable `sys.path` contiene el directorio actual, `PYTHONPATH`, y las predeterminadas dependencias de instalación.

### *Paquetes Python*

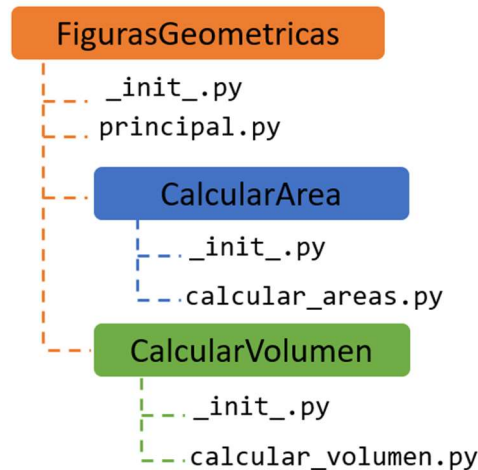
Los paquetes son una forma de estructurar el espacio de nombres de módulos de Python. Son directorios. Los paquetes pueden contener módulos y otros paquetes. El único requisito es que contengan un archivo llamado `__init__.py`.

El uso principal de `__init__.py` es inicializar los paquetes de Python, la inclusión del archivo `__init__.py` en un directorio indica al intérprete de Python que el directorio debe tratarse como un paquete de Python. El archivo `__init__.py` puede ser un archivo vacío. Una cosa común que se debe hacer en su `__init__.py` es importar las Clases, funciones, etc. seleccionadas al nivel del paquete para que puedan ser importadas convenientemente desde el paquete.

En el archivo `_init_.py` podemos definir la llamada a la lista `_all_`, a la cual podemos asignar la lista de todos los submódulos de nuestro paquete. Por ejemplo:

```
_all_ = [CalcularArea, CalcularVolumen,...]
```

Ahora, vamos a crear un paquete Figuras Geométricas desde el IDE PyCharm, con la siguiente estructura:



## Pasos:

1. Seleccione la ubicación dentro del proyecto, donde desee crear la estructura, con el botón derecho del mouse habilite el menú y seleccione New – Python Package, introduzca **FigurasGeometricas**. Debe aparecer la carpeta **FigurasGeometricas** y el archivo `_init_.py`
2. Desde la carpeta **FigurasGeometricas**, con el botón derecho del mouse habilite el menú y seleccione New – Python Package, introduzca **CalcularArea**. Debe aparecer la carpeta **CalcularArea** y el archivo `_init_.py`
3. Desde la carpeta **FigurasGeometricas**, con el botón derecho del mouse habilite el menú y seleccione New – Python Package, introduzca **CalcularVolumen**. Debe aparecer la carpeta **CalcularVolumen** y el archivo `_init_.py`

Ya hemos creado la estructura, ahora vamos a crear nuestros módulos:

1. Desde la carpeta **CalcularVolumen**, con el botón derecho del mouse habilite el menú y seleccione New – Python File, introduzca `calcular_volumen.py`. Copie el siguiente código:

```

"""
Módulo para el cálculo del volumen de diferentes
figuras geométricas
"""

```

```

def volumen_cubo(a):
    return a**3

```

2. Desde la carpeta **CalcularArea**, con el botón derecho del mouse habilite el menú y seleccione New – Python File, introduzca `calcular_area.py`. Copie el siguiente código:

```

"""

```

```
Módulo para el cálculo del área de diferentes
figuras geométricas
"""
```

```
# Librerías
```

```
import math
```

```
# Funciones para cálculo del área de figuras geométricas
```

```
def area_cuadrado(lado):
    a_cuadrado = lado**2
    return a_cuadrado
```

```
def area_triangulo(base, altura):
    a_triangulo = (base*altura)/2
    return a_triangulo
```

```
def area_circulo(radio):
    a_circulo = math.pi*radio**2
    return a_circulo
```

3. Desde la carpeta FigurasGeometricas, con el botón derecho del mouse habilite el menú y seleccione New – Python File, introduzca `principal.py`. Este será nuestro programa principal.

```
# Programa Principal Cálculo de Area y Volumen
```

```
from CalcularArea import calcular_areas
from CalcularVolumen import calcular_volumen
```

```
print("Cálculo del área de un cuadrado: {0} ".format(
    calcular_areas.area_cuadrado(int(input("Ingrese el
    tamaño del lado de cuadrado: ")))))
```

```
print("Cálculo del área de un círculo: {0} ".format(
    calcular_areas.area_circulo(int(input("Ingrese el
    radio del círculo: ")))))
```

```
print("El volumen de un cubo con a=40 es: ",
    calcular_volumen.volumen_cubo(40))
```

### *Sentencia from*

La sentencia `from` se utiliza en conjunto a la previa sentencia `import` para importar un módulo.

Algunos ejemplos:

```
from CalcularArea import calcular_areas
from CalcularVolumen import calcular_volumen
from CalcularArea.calcular_areas import area_cuadrado
from FigurasGeometricas.CalcularArea import calcular_areas
```





## Bibliografía

Wikipedia. (2 de Agosto de 2020). Obtenido de Wikipedia:  
<https://es.wikipedia.org/wiki/Factorial>

