

# IPI 2019 - Unidad TRES

UNNOBA



## Modularización

Hasta ahora los problemas vistos son sencillos. Pero en el mundo real los **problemas son más complejos y de mayor magnitud.**

Uno de los métodos más conocidos para resolver un problema complejo y grande es **dividirlo** en problemas más pequeños, llamados "subproblemas".

De esta manera, en lugar de resolver una tarea compleja y laboriosa, resolvemos otras más sencillas para que, a partir de ellas, se pueda llegar a la solución buscada originalmente.



### Parámetros

Los parámetros sirven la misión de comunicar a la función con quien la llama.

A los valores que se pasan al llamar a la función se los denotamos "argumentos".

1. Definir los parámetros que se pasan al llamar a la función y la misión que se realiza.
2. Definir la función que realiza la misión.
3. Definir la función que realiza la misión.

En la figura "Misión" se el parámetro formal y en la figura "Misión" el parámetro real o argumento.

### Funciones

¿Cómo modularizar en Python?

En Python contamos con **módulos y funciones**.

Comenzamos por las **funciones** sabiendo que contamos con **funciones** que poseen el propio lenguaje y los que **definen el usuario**.

Para **definir** una función en Python usamos el siguiente esquema:

```
def nombre_funcion(parametros):  
    # cuerpo de la función  
    return resultado
```

### Ámbito y alcance

El ámbito de una variable se define como el lugar en donde es variable y alcance puede ser definido.

En la figura "Misión" se el parámetro formal y en la figura "Misión" el parámetro real o argumento.



# IPI 2019 - Unidad TRES

UNNOBA



### Parámetros

Los parámetros tienen la misión de comunicar a la función con quien la invoca.  
A los valores que se pasan al invocar a la función se los denomina "argumentos".

```
1. def suma_uno(valor):  
    # Le suma 1 al argumento y lo retorna  
    return valor + 1 # Resultado de la función  
  
# Invocamos la función que definimos:  
2. >>> valor = suma_uno(5)
```

En la línea 1 "valor" es el **parámetro formal** y en la línea 2 el valor 5 es el **parámetro real o argumento**.

### Funciones

#### ¿Cómo modularizar en Python?

En Python contamos con **módulos** y **funciones**.

Comenzamos por las **funciones** sabiendo que contamos con las que provee el propio lenguaje y las que **definimos nosotros**.

Para **definir** una función en Python usamos el siguiente esquema:

```
def nombre_de_la_funcion(argumentos):  
    """ Documentación que describe qué hace la función """  
    # cuerpo de la función  
    return valor_de_retorno
```

El nombre de la función debe ser en minúsculas y puede estar precedido por un prefijo que indique su propósito (por ejemplo, `_privado` para funciones privadas).

## Modularización

Hasta ahora los problemas vistos son sencillos. Pero en el mundo real los **problemas son más complejos** y de **mayor magnitud**.

Uno de los métodos más conocidos para resolver un problema complejo y grande es **dividirlo** en problemas más pequeños, llamados "subproblemas".

De esta manera, en lugar de resolver una tarea compleja y laboriosa, resolvemos otras más sencillas para que, a partir de ellas, se pueda llegar a la solución buscada originalmente.



### Ámbito y alcance

El ámbito de una variable se define como el lugar en donde esa variable **sólo** puede ser utilizada.

Las **Locales** son creadas en las funciones. Mueren por las que mueren fuera de donde fueron creadas. **Globales**.

Las **variables locales** tienen una vida efímera, sólo existen durante el momento en el que se llama la función.

```
x = 10  
  
def suma(x, y):  
    # Variable local  
    z = x + y  
    return z  
  
print(suma(5, 3))  
print(x)  
print(z)
```

El ámbito de una variable se define como el lugar en donde esa variable sólo puede ser utilizada.

### Módulos

Un módulo es un archivo de Python que contiene una o más definiciones de funciones, clases, variables, etc.

Los módulos se importan en otros programas de Python para utilizarlos.

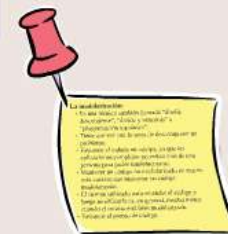
El nombre de un módulo se convierte en un nombre de módulo al importarlo.

```
import math  
print(math.sqrt(16))
```

El nombre de un módulo se convierte en un nombre de módulo al importarlo.

# IPI 2019 - Unidad TRES

UNNOBA



# Modularización

Hasta ahora los problemas vistos son sencillos. Pero en el mundo real los **problemas** son **más complejos** y de **mayor magnitud**.

Uno de los métodos más conocidos para resolver un problema complejo y grande es **dividirlo** en problemas más pequeños, llamados “subproblemas”.

De esta manera, en lugar de resolver una tarea compleja y laboriosa, resolvemos otras más sencillas para que, a partir de ellas, se pueda llegar a la solución buscada originalmente.

Supongamos el siguiente dibujo



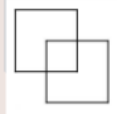
Si observamos con detalle la figura vemos como están formados por muchos cuadrados. Pero como están muy juntos y son muy sencillos que se repiten, ¿cómo dibujar un cuadrado?

Dibujamos un cuadrado



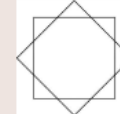
Definitivamente “dibujar un cuadrado” es una tarea mucho más sencilla que la propuesta inicialmente.

Dibujamos varios cuadrados



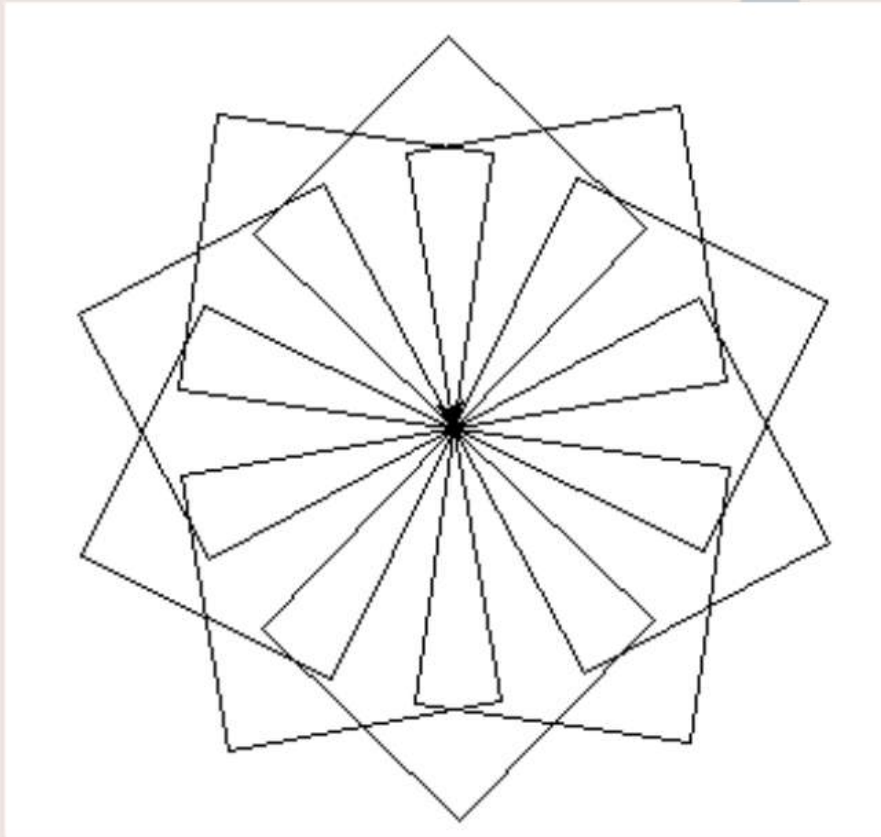
Para resolver el problema los cuadrados se están dibujando como la respuesta, lo único que cambia es: En primer orden se dibuja los cuadrados para formar la figura más compleja. Podemos pensar entonces en los pasos sencillos:  
- “Dibujar un cuadrado”  
- “Rotar o girar a la derecha la figura”

Rotar o girar...



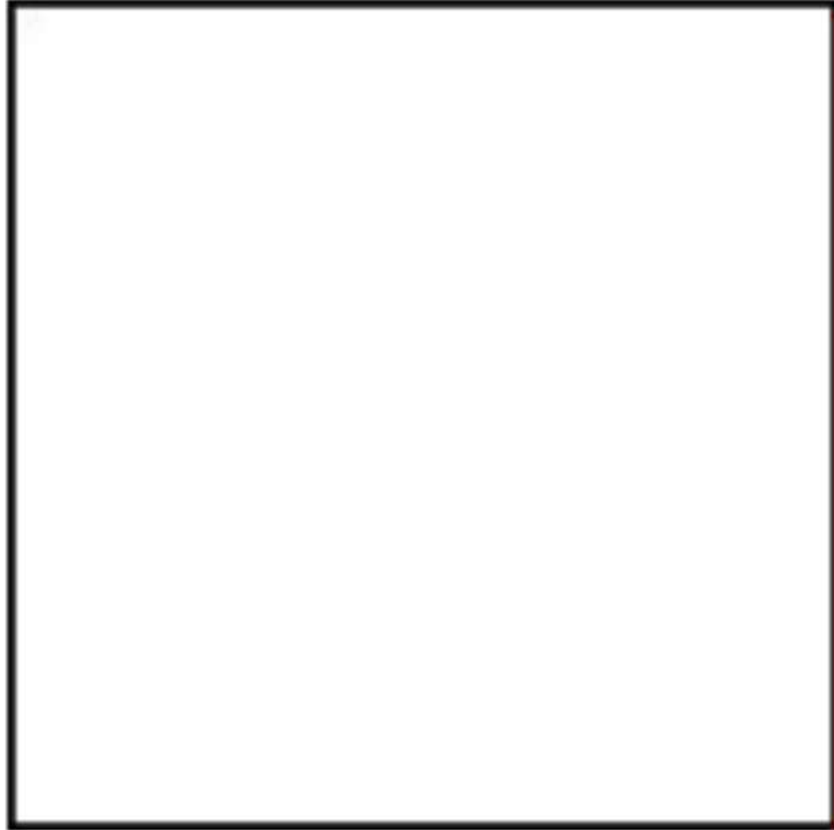
Entonces podemos dividir el problema inicial en los siguientes pasos:  
- “Dibujar un cuadrado”  
- “Rotar o girar a la derecha”  
De tal manera que llegaremos a la figura inicial.

# Supongamos el siguiente dibujo



Si observamos con detalle la figura vamos a notar que se trata de varios cuadrados. Entonces encontramos una tarea más sencilla que es resolver “cómo dibujar un cuadrado”.

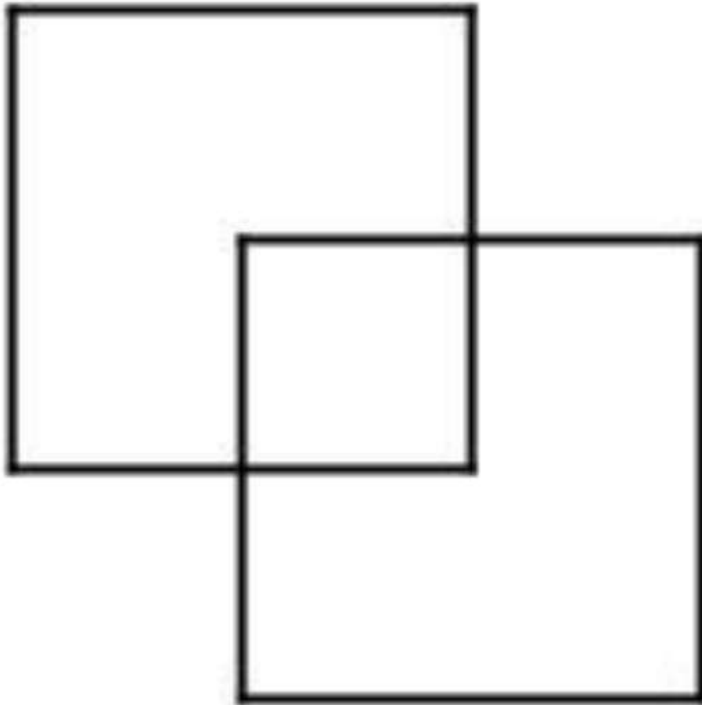
# Dibujamos un cuadrado



Definitivamente "dibujar un cuadrado" es una tarea mucho más sencilla que la propuesta inicialmente.



# Dibujamos varios cuadrados

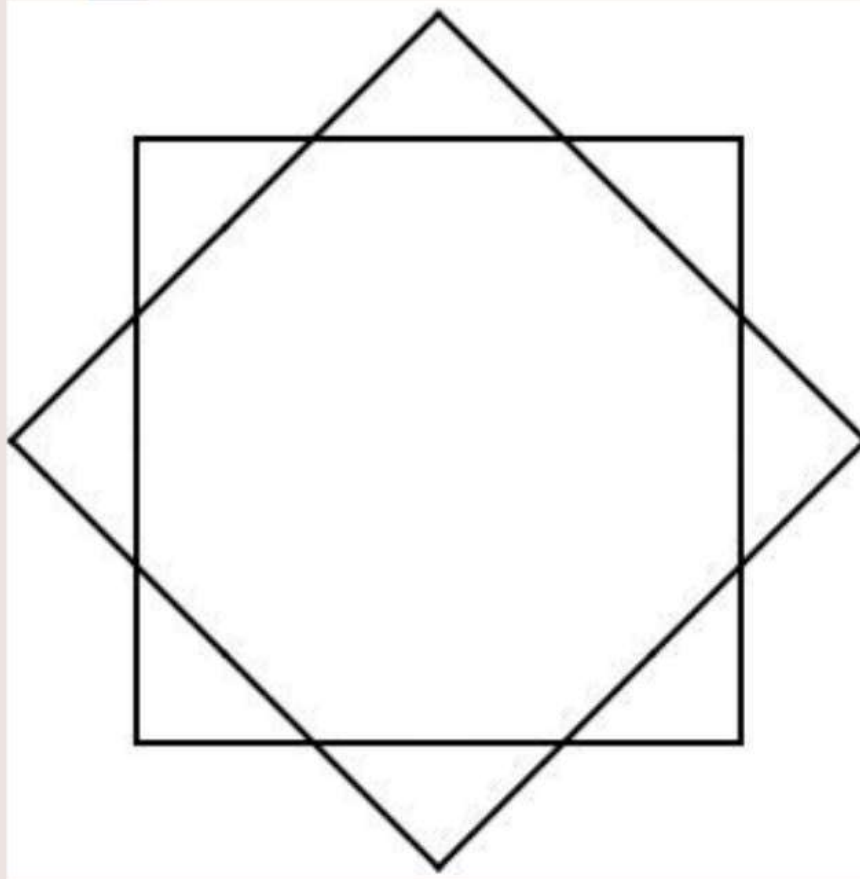


Pero tenemos un problema: los cuadrados no están ubicados como se espera, habría que rotarlos. Entonces debemos ubicar los cuadrados para formar la figura más compleja.

Podemos pensar entonces en dos tareas sencillas:

- “dibujar un cuadrado”
- “rotar o girar a la derecha la figura”.

# Rotar o girar...

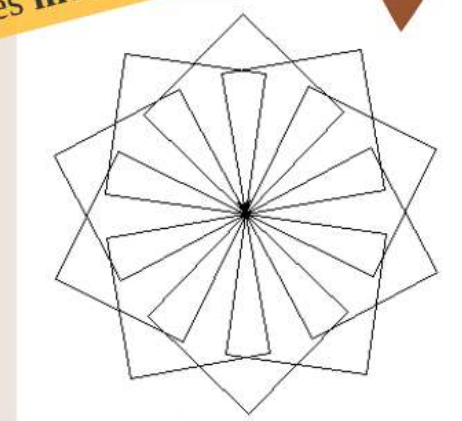


Luego podríamos dividir el problema inicial en las siguientes tareas:

- “dibujar cuadrado”
- “rotar el cuadrado un grado”

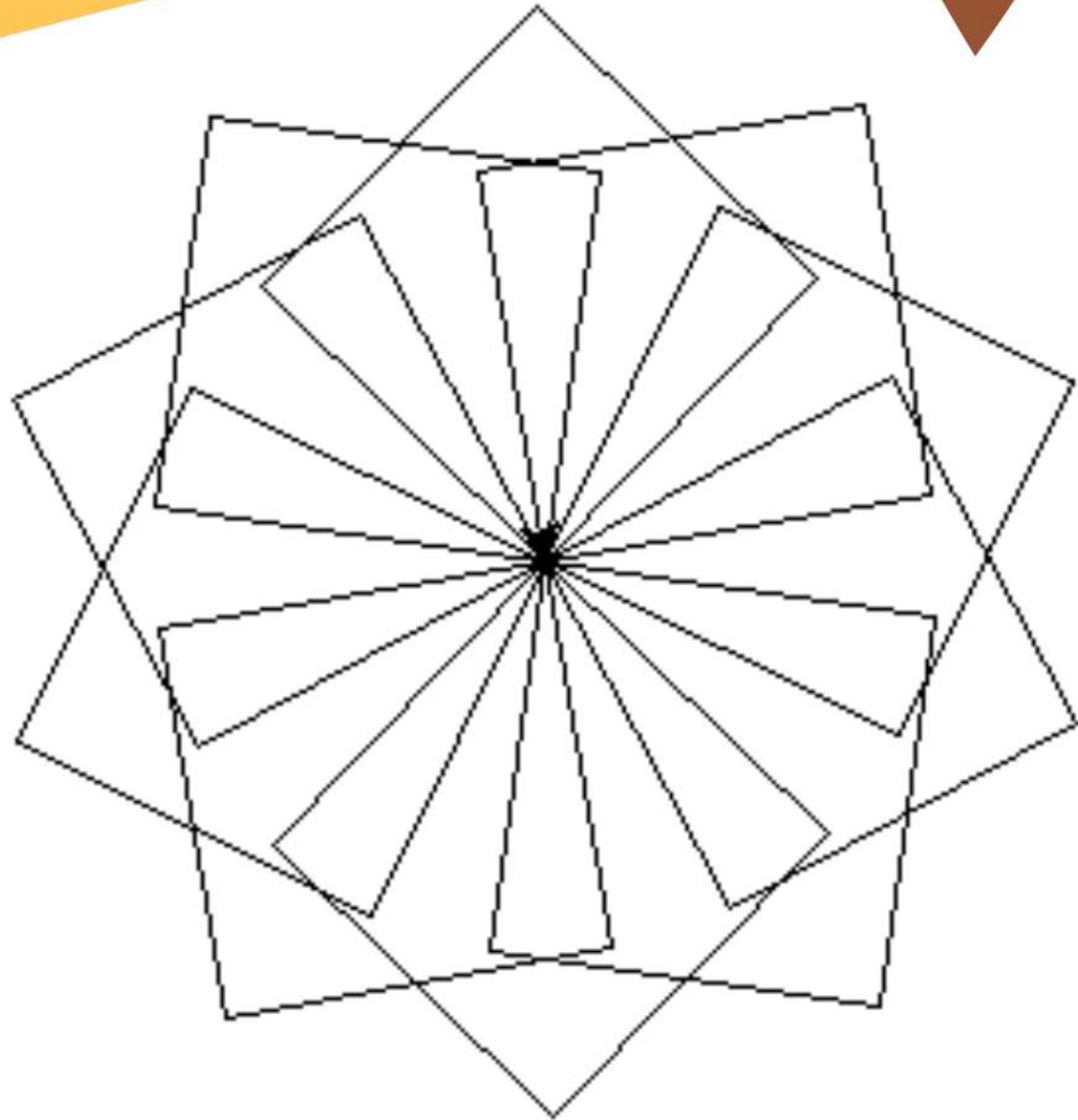
De tal manera que lleguemos a la figura inicial:

Esto es **modularizar**





Esto es **modularizar**





### **La modularización:**

- Es una técnica también llamada “diseño descendente”, “divide y vencerás” o “programación top-down”.
- Tiene que ver con la tarea de descomponer un problema.
- Favorece el trabajo en equipo, ya que las aplicaciones complejas necesitan más de una persona para poder implementarse.
- Mantener un código no modularizado es mucho más costoso que mantener un código modularizado.
- El tiempo utilizado para entender el código y luego modificarlo es, en general, mucho menor cuando el mismo está bien modularizado.
- Favorece el pseuso de código.

## **La modularización:**

- Es una técnica también llamada “diseño descendente”, “divide y vencerás” o “programación top-down”.
- Tiene que ver con la tarea de descomponer un problema.
- Favorece el trabajo en equipo, ya que las aplicaciones complejas necesitan más de una persona para poder implementarse.
- Mantener un código no modularizado es mucho más costoso que mantener un código modularizado.
- El tiempo utilizado para entender el código y luego modificarlo es, en general, mucho menor cuando el mismo está bien modularizado.
- Favorece el pseuso de código.

# Funciones

## ¿Cómo modularizar en Python?

En Python contamos con **módulos** y **funciones**.

Comencemos por las **funciones** sabiendo que contamos con las que provee el propio lenguaje y las que **define el usuario**.

Para **definir** una función en Python usamos el siguiente esquema:

Para indicar que estamos escribiendo el cuerpo de la función debemos dejar espacios (en la sangría). Se recomienda que sean cuatro espacios.

El "valor\_de\_retorno" es opcional. en el caso de no existir la sentencia "return", la función devolverá el valor por defecto: "none".

```
def nombre_de_la_función(parámetros):  
    """Documentación que describe que hace la función"""  
    Instrucción 1  
    Instrucción 2  
    ...  
    return valor_de_retorno
```

Usamos la palabra clave "def" para indicarle a Python que estamos definiendo una nueva función. No debemos dejar sangrías en este primer renglón y terminamos con ":"

Luego, en la primera línea de una función podemos agregar una cadena de caracteres describiendo qué hace la función, qué valores toman los parámetros, qué devuelve, etc. Normalmente se utiliza una cadena de caracteres multilinea (usando triple comillas) para poder escribir la documentación más fácilmente.

**Invocación**

```
def suma(a, b):  
    """Función que suma dos números"""  
    return a + b
```

Para invocar la función "suma" se usa la siguiente sintaxis:

```
resultado = suma(2, 3)
```

**Funciones incorporadas**

Python incluye muchas funciones incorporadas, como `print()`, `len()`, `input()`, etc.

```
print("Hola mundo")
```

En Python contamos con **módulos** y **funciones**.

Comencemos por las **funciones** sabiendo que contamos con las que provee el propio lenguaje y las que **define el usuario**.

Para **definir** una función en Python usamos el siguiente esquema:

Para indicar que estamos escribiendo el cuerpo de la función debemos dejar espacios (en la sangría). Se recomienda que sean cuatro espacios.

El "valor\_de\_retorno" es opcional. en el caso de no existir la sentencia "return", la función devolverá el valor por defecto: "none".

```
def nombre_de_la_función(parámetros):  
    """Documentación que describe que hace la función"""  
    Instrucción 1  
    Instrucción 2  
    ...  
    return valor_de_retorno
```

Usamos la palabra clave "def" para indicarle a Python que estamos definiendo una nueva función. No debemos dejar sangrías en este primer renglón y terminamos con ":"

Luego, en la primer línea de una función podemos agregar una cadena de caracteres describiendo qué hace la función, qué valores toman los parámetros, qué devuelve, etc. Normalmente se utiliza una cadena de caracteres multilínea (usando triple comillas) para poder escribir la documentación más fácilmente.

```
def suma(a, b):  
    """Función que suma dos números"""  
    return a + b
```

Funciones incorporadas

```
def print():  
    """Función incorporada para imprimir en pantalla"""  
    pass
```

El lenguaje Python incluye una gran cantidad de funciones incorporadas que se pueden utilizar directamente sin necesidad de definir una función.



# Valor de retorno

- Toda función **retorna un valor** como resultado de su invocación.
- El valor retornado permite **comunicar** a la función con la parte del programa que la invoca.
- Para retornar un valor, se utiliza la sentencia **return** seguida del valor a retornar.
- Tanto la sentencia **return** como el valor a retornar pueden omitirse.

Vamos a escribir entonces la función **suma\_uno()**:

```
def suma_uno(valor):  
    # Suma 1 al argumento y lo retorna  
    return valor + 1
```

Volvamos a escribir la función:

```
def suma_uno(valor):  
    # Suma 1 al argumento y lo retorna  
    return valor + 1
```

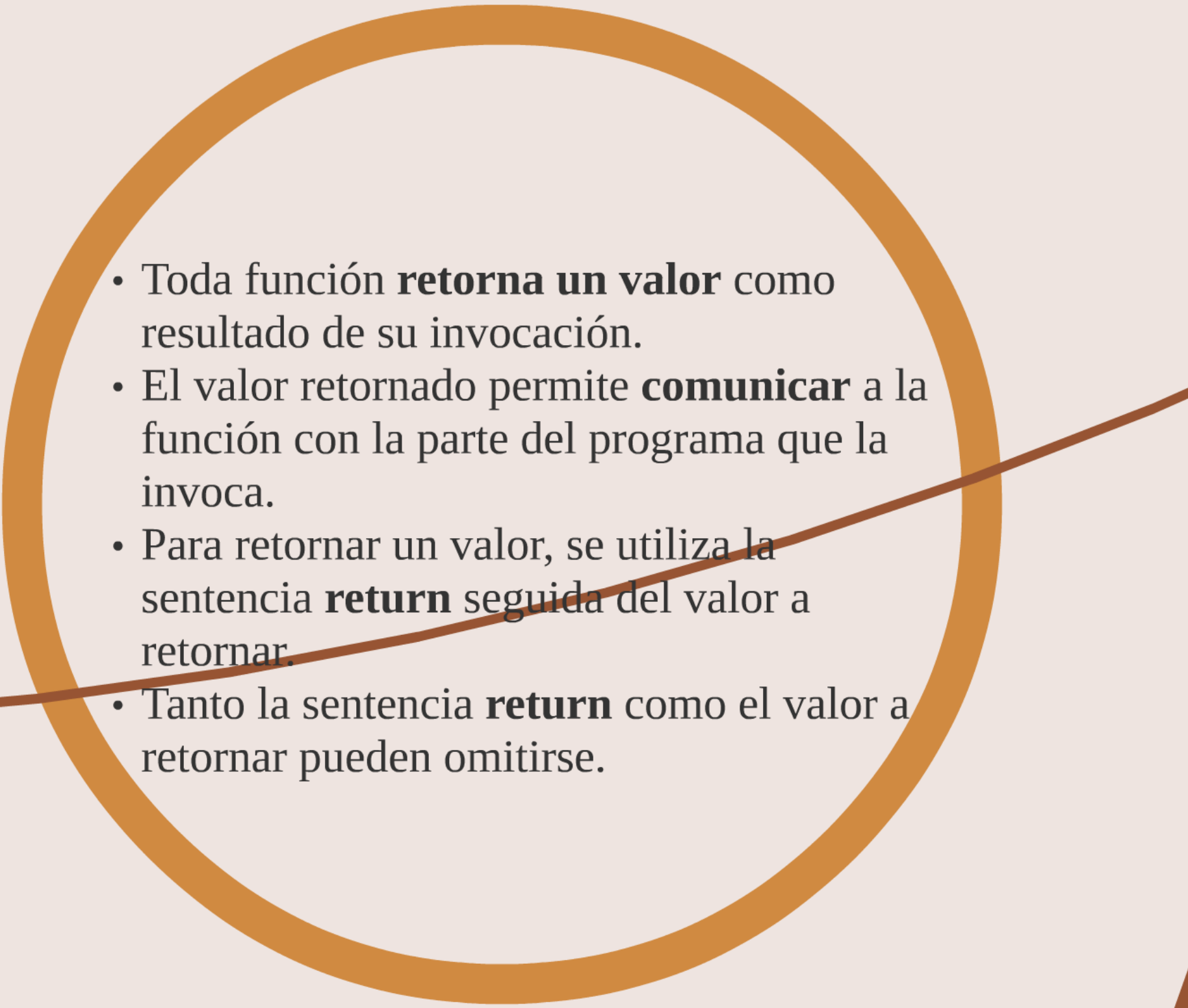
Evitemos entonces a las funciones que no retornan valor alguno o las llamamos para efectos de prueba. En Python, no se hace distinción alguna.

¿Qué sucede acá?

```
def bienvenida():  
    # Muestra un mensaje de bienvenida  
    print("¡Hola mundo!")
```

```
>>> valor = bienvenida()  
>>> valor  
>>> print(valor)  
None
```



- 
- A large orange circle is centered on the slide. A thin orange line enters from the right side and extends diagonally across the slide, passing behind the text.
- Toda función **retorna un valor** como resultado de su invocación.
  - El valor retornado permite **comunicar** a la función con la parte del programa que la invoca.
  - Para retornar un valor, se utiliza la sentencia **return** seguida del valor a retornar.
  - Tanto la sentencia **return** como el valor a retornar pueden omitirse.

Vamos a escribir entonces la función **suma\_uno()**:

```
def suma_uno(valor):  
#Le suma 1 al argumento y lo imprime  
print(valor + 1)
```

*Esta función, al no tener  
sentencia "return",  
retornará el valor "none" luego  
de ser invocada.*

Volvamos a escribir la función:

```
def suma_uno(valor):  
#Le suma 1 al argumento y lo retorna  
return(valor + 1)
```

*Esta versión, en lugar de  
imprimir el resultado obtenido,  
lo va a retornar.*

*En otros lenguajes, a las función que no retornan  
valor alguno se las denomina procedimientos.  
En Python, no se hace distinción alguna.*

Vamos a escribir entonces la función **suma\_uno()**:

```
def suma_uno(valor):  
    Le suma 1 al argumento y lo imprime  
    print(valor + 1)
```

*Esta función, al no tener sentencia "return", retornará el valor "none" luego de ser invocada.*

Volvamos a escribir la función:

```
def suma_uno(valor):  
    Le suma 1 al argumento y lo retorna
```

*Esta versión, en lugar de*

```
def suma_uno(valor):  
#Le suma 1 al argumento y lo imprime  
    print(valor + 1)
```

Esta función, al no tener  
sentencia "return",  
retornará el valor "none" luego  
de ser invocada.

## Volvamos a escribir la función:

```
def suma_uno(valor):  
#Le suma 1 al argumento y lo retorna  
    return(valor + 1)
```

Esta versión, en lugar de  
imprimir el resultado obtenido,  
lo va a retornar:

En otros lenguajes, a las función que no retornan  
valor alguno se las denomina procedimientos.  
En Python, no se hace distinción.

valor):  
argumento y lo retorna  
+ 1)

En otros lenguajes, a las función que no retornan valor alguno se las denomina procedimientos. En Python, no se hace distinción alguna.

## *¿Qué sucede acá?*

```
def bienvenida():  
    #Muestra un mensaje de Bienvenida  
    print("¡Hola mundo!")
```

.....

```
>>> valor = bienvenida()  
¡Hola mundo!  
>>> valor  
>>> print(valor)  
None
```



# *Invocación*

Una vez definida una función, puede invocarse. Veamos un ejemplo:

```
def máximo(x, y):  
    #Retorna x si x es mayor que y, sino retorna y  
    if y < x:  
        return x  
    else:  
        return y
```

Ahora para que la función "trabaje" hay que invocarla:

```
>>> máximo(5, 10)  
10  
>>> máximo(15, -5)  
15
```

# *Funciones incorporadas*

Python trae muchas funciones incorporadas, entre ellas `print()`, `input()`, `abs()` y `len()`.

```
>>> abs(-5)
5
>>> s = "Una cadena de texto"
>>> len(s)
19
```

Es conveniente reutilizar esos trozos de código con comportamiento definidos y sólo definir aquellas nuevas funciones que si hagan falta.

# Parámetros

Los parámetros tienen la misión de comunicar a la función con quien la invoca.

A los valores que se pasan al invocar a la función se los denomina “argumentos”.

```
1. def suma_uno(valor):  
    #Le suma 1 al argumento y lo retorna  
    return valor + 1 # Resultado de la función  
  
# Invocamos la función que definimos:  
2. >>> valor = suma_uno(5)
```

En la línea 1 “valor” es el **parámetro formal** y en la línea 2 el valor 5 es el **parámetro real o argumento**.

Los parámetros reales y las funciones se asocian por nombre y se crean los nombres. Dependiendo del número de valores se pasan, el primer argumento se asocia al primer parámetro formal, el segundo argumento al segundo parámetro formal, y así siguiendo para todos los que resten.

Por ejemplo:

def suma_uno(x, y): return x + y + 1 # Invocamos la función que definimos: suma_uno(5, 7)	La función "suma_uno" requiere dos parámetros formales: "x" y "y". Luego se invocó a la función "suma_uno" con los argumentos "5" y "7".
def suma_uno(x, y): return x + y + 1 # Invocamos la función que definimos: suma_uno(5, 7)	La función "suma_uno" requiere dos parámetros formales: "x" y "y". Luego se invocó a la función "suma_uno" con los argumentos "5" y "7".

Los parámetros reales y los formales se asocian por posición y no por su nombre. Independientemente de cómo se llamen, el primer argumento se asocia al primer parámetro formal, el segundo argumento al segundo parámetro formal, y así siguiendo para todos los que restan.

### Por ejemplo:

```
def concatenar(x, y, z):  
    return x + y + z  
.....  
# Invocamos la función que definimos:  
concatenar('a', 'b', 'c')  
'abc'
```

La función “concatenar” tiene tres **parámetros formales**: “x”, “y”, “z”. Luego la invocamos con los **argumentos** “a”, “b” y “c”.

Como la asociación es por posición, “x” toma el valor de “a”, “y” toma el valor de “b”, y “z” toma el valor de “c”.

Hasta el momento hablamos de **funciones**, como una manera de **modularizar** en Python. Pero también tenemos a los **MÓDULOS**

# Módulos

Al salir del intérprete de Python, las definiciones realizadas (funciones y variables) se pierden.

Por lo tanto, si se quiere escribir un programa más o menos largo, es mejor crear un *guión*, o *script*.

Las definiciones de un módulo pueden ser importadas a otros módulos o al módulo principal.

Un módulo es un archivo que tiene como nombre el nombre del módulo con extensión `.py`.

Luego para importar el módulo usamos  
`>>> from nombre_del_archivo import *`

También es probable que se quiera usar una función útil desde distintos programas sin copiar su definición en cada uno de ellos.

Para soportar lo anterior, Python tiene una manera de poner definiciones en un archivo y usarlas en un script o en una instancia interactiva del intérprete.

Tal archivo es llamado **MÓDULO**.



es de **funciones**, como una manera de  
y unon. Pero también tenemos a los **MÓDULOS**

# Módulos

Al salir del intérprete de Python, las definiciones realizadas (funciones y variables) se pierden.

Por lo tanto, si se quiere escribir un programa más o menos largo, es mejor crear un *guión*, o *script*.

Las definiciones de un módulo pueden ser importadas a otros módulos o al módulo principal.

Un módulo es un archivo que tiene como nombre el nombre del módulo con extensión *.py*.

Luego para importar el modulo usamos  
>>> **from nombre\_del\_archivo import \***

También es probable que se quiera usar una función útil desde distintos programas sin copiar su definición en cada uno de ellos.

Para soportar lo anterior, Python tiene una manera de poner definiciones en un archivo y usarlas en un script o en una instancia interactiva del intérprete.

Tal archivo es llamado MÓDULO.

Veamos un ejemplo de una herramienta de dibujo con comandos.

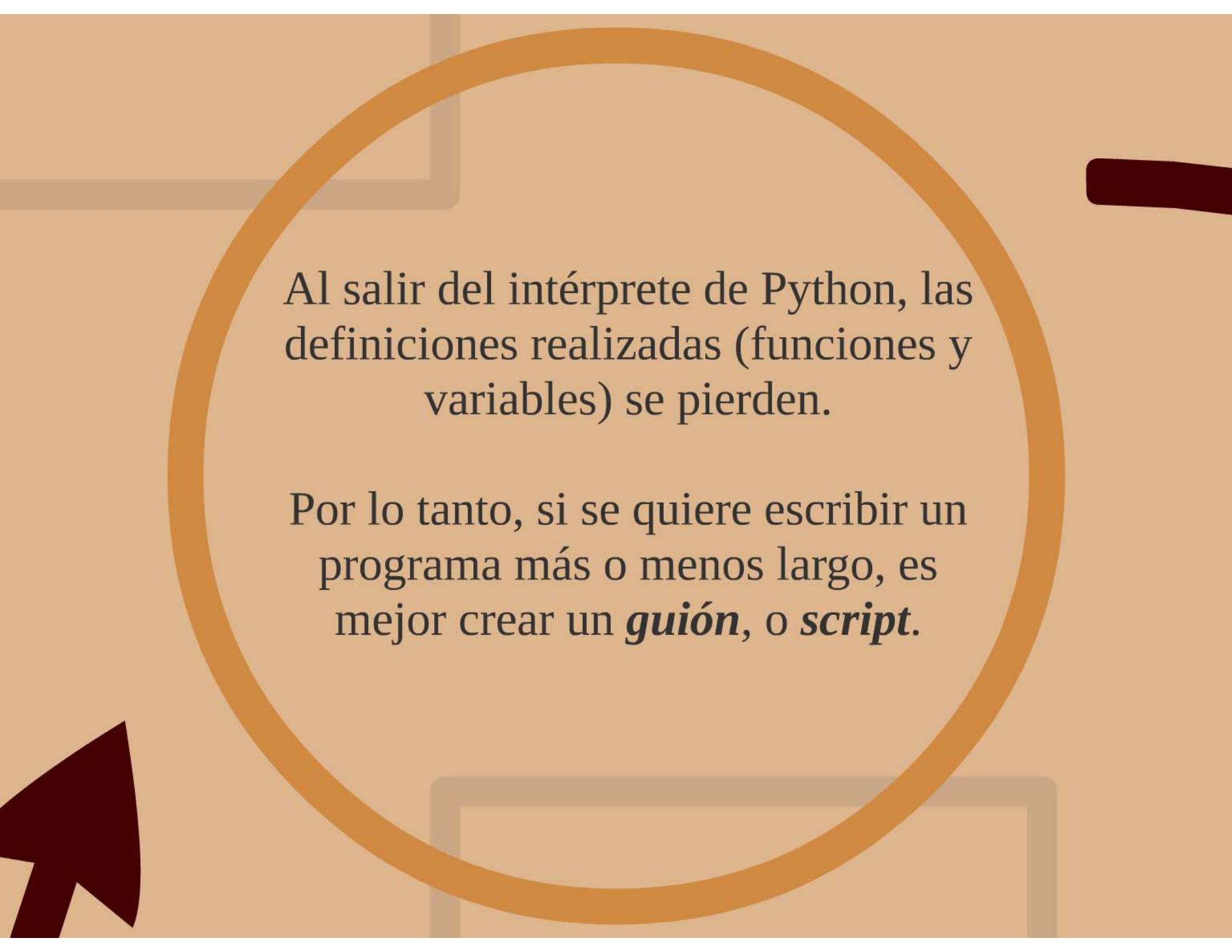
```
from turtle import *\nfor i in range(3):\n    begin_fill()\n    fillcolor("red")\n    circle(50)\n    end_fill()\n    penup()\n    forward(100)\n    pendown()
```



Hasta el momento hablamos de **funciones**, como una manera de **modularizar** en Python. Pero también tenemos a los **MÓDULOS**

Al salir del intérprete de Python, las definiciones realizadas (funciones, variables) se pierden.  
Por lo tanto, si se quiere escribir un programa más o menos largo, lo mejor es crear un **guion**, o sea, un archivo de texto con extensión .py.

Mód



Al salir del intérprete de Python, las definiciones realizadas (funciones y variables) se pierden.

Por lo tanto, si se quiere escribir un programa más o menos largo, es mejor crear un ***guión***, o ***script***.

También es probable que se quiera usar una función útil desde distintos programas sin copiar su definición en cada uno de ellos.

Para soportar lo anterior, Python tiene una manera de poner definiciones en un archivo y usarlas en un script o en una instancia interactiva del intérprete.

Tal archivo es llamado **MÓDULO**.

Las definiciones de un módulo pueden ser importadas a otros módulos o al módulo principal.

Un módulo es un archivo que tiene como nombre el nombre del módulo con extensión **.py**.

Luego para importar el modulo usamos  
**>>> from nombre\_del\_archivo import \***



# Turtle

Veamos un ejemplo con el módulo **Turtle** de Python que es una excelente herramienta de dibujo. Turtle permite dibujar mediante instrucciones o comandos.

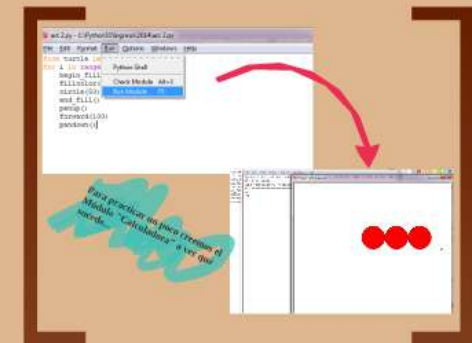
```
from turtle import*  
for i in range(3):  
    begin_fill()  
    fillcolor("red")  
    circle(50)  
    end_fill()  
    penup()  
    forward(100)  
    pendown()
```

Con esta línea se indica que se quiere importar desde "turtle" las variables, funciones y tipos (entre otras cosas) para ser usadas.

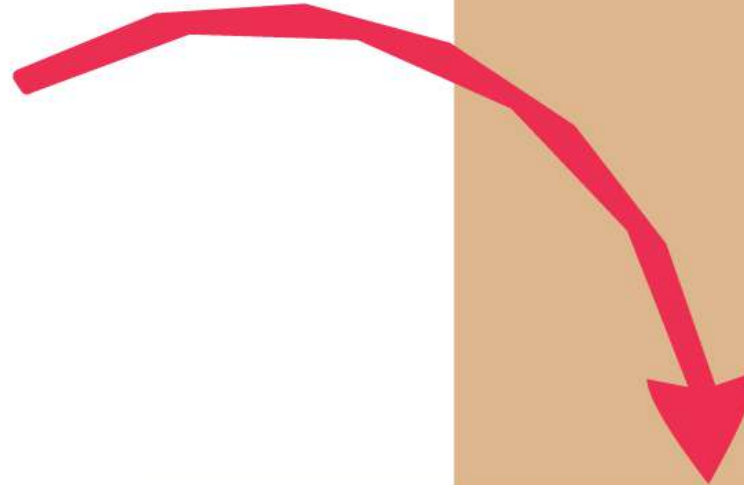
**begin\_fill()**, indica que las figuras cerradas que se describan a continuación deberán ser pintadas con el color especificado por **fillcolor(color)**,

**fillcolor(color)**, define el color de relleno de figuras cerradas

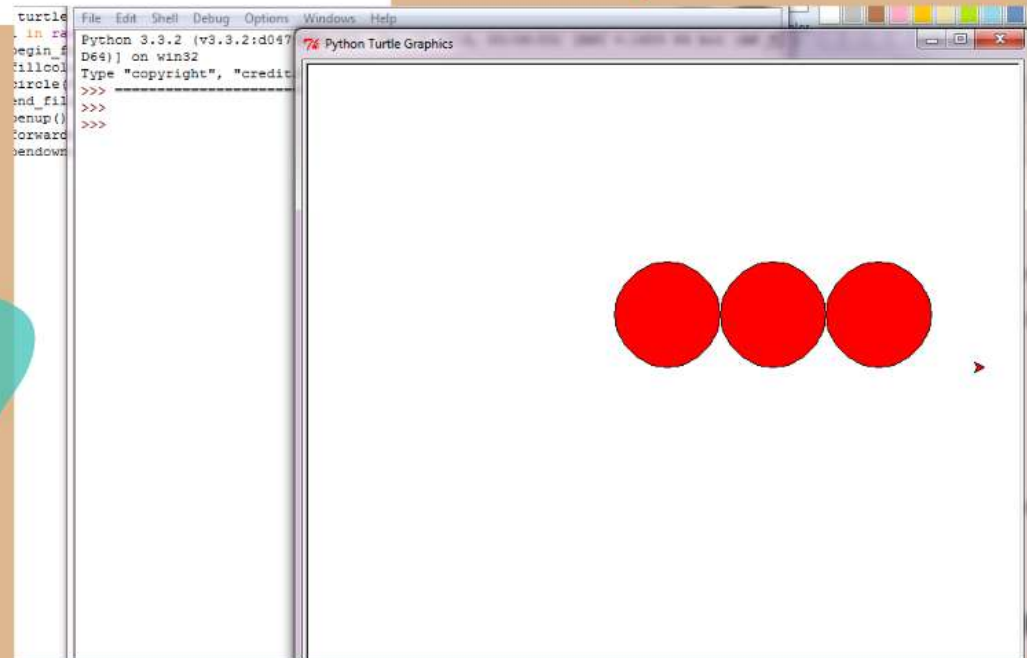
**end\_fill()**, pinta todas las figuras cerradas dibujadas a partir de **begin\_fill()**



```
7% act 2.py - C:\Python33\Ingreso\2014\act 2.py
File Edit Format Run Options Windows Help
from turtle import *
for i in range(3):
    begin_fill()
    fillcolor("red")
    circle(50)
    end_fill()
    penup()
    forward(100)
    pendown()
```



Para practicar un poco creemos el  
Módulo "Calculadora" a ver qué  
sucede...





**Para practicar un poco creemos el  
Módulo "Calculadora" a ver qué  
sucede...**

```
circle  
end_fil >>>  
penup() >>>  
forward  
pendown
```

# Ámbito y alcance

El ámbito de una variable se define como el lugar en donde esa variable solamente puede ser utilizada.

Las **Locales** son creadas en las funciones. Mientras que las que residen fuera de toda función se denominan **Globales**.

Las **variables locales** tienen una vida efímera, sólo existen durante el momento en el que es llamada la función.

```
Ámbito global
x = 10

def func():
    x = 20
    return x
    Ámbito local a func()

print(x)
print(func())
print(x)
```

el ámbito de una variable local está limitado al código de la función en la que está definida



```
def cajanegra():
    c = 3
    print('La variable c dentro de la función tiene por valor', c)
```

```
a = 1
b = 2
c = 5
```

```
print(a)
print(b)
cajanegra()
print('La variable c fuera de la función tiene por valor', c)
```

El resultado de la ejecución sería:

```
>>>
1
2
La variable c dentro de la función
tiene por valor 3
La variable c fuera de la función
tiene por valor 5
```



Ámbito global

```
x = 10
```

```
def func():
```

```
    x = 20
```

```
    return x
```

```
print(x)
```

```
print(func())
```

```
print(x)
```

Ámbito local  
a func()

el ámbito de una variable local está limitado al código de la función en la que está definida

durante el momento en el que es llamada la función.

```
def cajanegra():  
    c = 3  
    print('La variable c dentro de la función tiene por valor', c)
```

de una  
l está  
digo de  
la que

```
a = 1  
b = 2  
c = 5
```

```
print(a)  
print(b)  
cajanegra()  
print('La variable c fuera de la función tiene por valor', c)
```

**El resultado de la ejecución sería:**

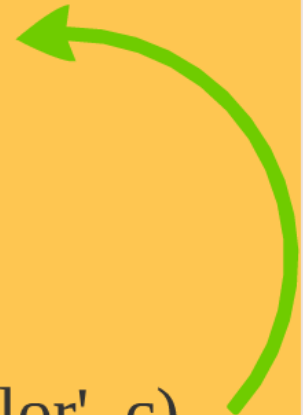
>>>

1

2

La variable c dentro de la función  
tiene por valor 3

La variable c fuera de la función  
tiene por valor 5





# Globales

Las variables **globales** tienen la particularidad de que también son visibles dentro de la función:

Todo intento de modificar una variable global desde dentro de una función mediante una nueva asignación fracasa

def cajanegra():

c = 3

print(a)

print(b)

print(c)

a = 1

b = 2

cajanegra()

El resultado de la ejecución sería:

```
>>>
1
2
3
```

def cajanegra():

c = 3

print(a)

b = 5

print('La variable b dentro de la función vale', b)

print(c)

a = 1

b = 2

cajanegra()

print('La variable b fuera de la función sigue valiendo', b)

El resultado de la ejecución sería:

```
>>> cajanegra()
1
La variable b dentro de la función
vale 5
3
```

El resultado de la ejecución sería:

La variable b fuera de la función sigue valiendo 2

Dentro de una función, la misma variable no puede ser en unos momentos global y en otros local. Si hay una asignación, aunque sea posterior a su uso como variable global, la variable será considerada local y se producirá un error:

def cajanegra():

c = 3

print(b)

b = 5

a = 1

b = 2

cajanegra()

la variable **b** ha sido referenciada antes de haber recibido una asignación, algo prohibido en Python. La asignación hace que **b** se trate como local, invalidando su rol global.



def cajanegra():

c = 3

global b

b = 5

print('Dentro de la función b vale', b)

a = 1

b = 2

cajanegra()

print('Fuera de la función b también vale', b)

Valdrá 5 ambas veces

Puede resultar conveniente, dentro de una función, cambiar el valor de una variable global mediante una nueva asignación. Para lograr esto, hay que calificar la variable externa dentro de la función empleando la palabra global.



# *Globales*

Las variables **globales** tienen la particularidad de que también son visibles dentro de la función:

Todo intento de modificar una variable global desde dentro de una función mediante una nueva asignación fracasa

```
def
```

```
c
```

```
E
```

```
E
```

```
a =
```

```
b =
```

```
caj
```

```
def cainogra():
```

tienen la  
también son

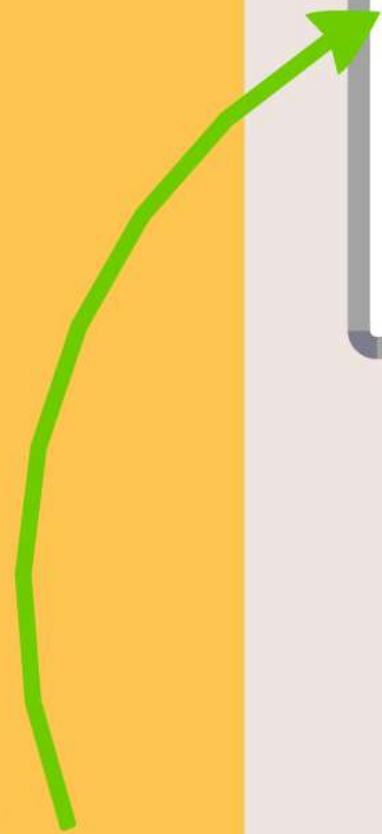
una  
e una  
nueva

```
def cajanegra():  
    c = 3  
    print(a)  
    print(b)  
    print(c)
```

```
a = 1  
b = 2  
cajanegra()
```

El resultado de la ejecución

```
>>>  
1  
2  
3
```



variable global desde dentro de una función mediante una nueva asignación fracasa

D = 2  
cajanegra()

```
def cajanegra():  
    c = 3  
    print(a)  
    b = 5  
    print('La variable b dentro de la función vale', b)  
    print(c)  
  
a = 1  
b = 2  
cajanegra()  
print('La variable b fuera de la función sigue valiendo',  
b)
```

después de esta línea

El resultado de la ejecución sería:

```
>>> cajanegra()  
1  
La variable b dentro de la función  
vale 5  
3
```

después de esta línea

El resultado de la ejecución sería:

La variable b fuera de la función  
sigue valiendo 2

Dentro de  
global y  
como va  
error:

```
def caja  
    c = 3  
    print(  
    b = 5
```

```
a = 1  
b = 2  
cajaneg
```

Puede res  
función,  
global me  
lograr es  
externa d  
palabra gl

después de esta línea



# El resultado de la ejecución sería:

```
>>> cajanegra()
```

```
1
```

```
La variable b dentro de la función  
vale 5
```

```
3
```

variable global desde dentro de una función mediante una nueva asignación fracasa

D = 2  
cajanegra()

```
def cajanegra():  
    c = 3  
    print(a)  
    b = 5  
    print('La variable b dentro de la función vale', b)  
    print(c)  
  
a = 1  
b = 2  
cajanegra()  
print('La variable b fuera de la función sigue valiendo',  
b)
```

después de esta línea

El resultado de la ejecución sería:

```
>>> cajanegra()  
1  
La variable b dentro de la función  
vale 5  
3
```

después de esta línea

El resultado de la ejecución sería:

La variable b fuera de la función  
sigue valiendo 2

Dentro de  
global y  
como va  
error:

```
def cajanegra():  
    c = 3  
    print(a)  
    b = 5
```

```
a = 1  
b = 2  
cajanegra()
```

Puede res  
función,  
global me  
lograr es  
externa d  
palabra gl



ción

después de esta línea

ción

**El res**

La variación  
sigue vali

**El resultado de la ejecución sería:**

La variable b fuera de la función  
sigue valiendo 2

ularidad de que también  
es dentro de la función:

**Todo intento de modificar una  
variable global desde dentro de una  
función mediante una nueva  
asignación fracasa**

janegra():

3

variable global desde dentro de una función mediante una nueva asignación fracasa

D = 2  
cajanegra()

```
def cajanegra():  
    c = 3  
    print(a)  
    b = 5  
    print('La variable b dentro de la función vale', b)  
    print(c)  
  
a = 1  
b = 2  
cajanegra()  
print('La variable b fuera de la función sigue valiendo',  
b)
```

después de esta línea

El resultado de la ejecución sería:

```
>>> cajanegra()  
1  
La variable b dentro de la función  
vale 5  
3
```

después de esta línea

El resultado de la ejecución sería:

La variable b fuera de la función  
sigue valiendo 2

Dentro de  
global y  
como va  
error:

```
def cajanegra():  
    c = 3  
    print(a)  
    b = 5
```

```
a = 1  
b = 2  
cajanegra()
```

Puede res  
función,  
global me  
lograr es  
externa d  
palabra gl

Dentro de una función, la misma variable no puede ser en unos momentos global y en otros local. Si hay una asignación, aunque sea posterior a su uso como variable global, la variable será considerada local y se producirá un error:

```
def cajanegra():
```

```
    c = 3
```

```
    print(b)
```

```
    b = 5
```

Por esto Python informará un error

```
a = 1
```

```
b = 2
```

```
cajanegra()
```

la variable **b** ha sido referenciada antes de haber recibido una asignación, algo prohibido en Python. La asignación hace que b se trate como local, invalidando su rol global.



```
def cajanegra():
```

```
    c = 3
```

```
    global b
```

```
    b = 5
```

```
    print('Dentro de la función b vale', b)
```

```
a = 1
```

```
b = 2
```

```
cajanegra()
```

```
print('Fuera de la función b también vale', b)
```

El resultado de la ejecución sería:  
Dentro de la función b vale 5

El resultado de la ejecución sería:  
Fuera de la función b también vale 5

**Valdrá 5 ambas veces**

Puede resultar conveniente, dentro de una función, cambiar el valor de una variable global mediante una nueva asignación. Para lograr esto, hay que calificar la variable externa dentro de la función empleando la palabra global.

Dentro de una función, la misma variable no puede ser en unos momentos global y en otros local. Si hay una asignación, aunque sea posterior a su uso como variable global, la variable será considerada local y se producirá un error:

```
def cajanegra():
```

```
    c = 3
```

```
    print(b)
```

```
    b = 5
```

```
a = 1
```

```
b = 2
```

```
cajanegra()
```

Por esto Python informará un error

la variable **b** ha sido referenciada antes de haber recibido una asignación, algo prohibido en Python. La asignación hace que b se trate como local, invalidando su rol global.



```
def cajanegra():
```

```
    c = 3
```

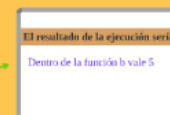
```
    global b
```

```
    b = 5
```

```
    print('Dentro de la función b vale', b)
```

```
a = 1
```

```
b = 2
```



Puede resultar conveniente, dentro de una



error.

```
def cajanegra():  
    c = 3  
    print(b)  
    b = 5
```

```
a = 1  
b = 2  
cajanegra()
```

Por esto Python informará un error

la variable  
una asigna  
La asigna  
invalidar

```
a = 1  
b = 2  
cajanegra()
```

Python informará un error



Puede resultar conveniente, dentro de una función, cambiar el valor de una variable global mediante una nueva asignación. Para lograr esto, hay que calificar la variable externa dentro de la función empleando la palabra `global`.

def  
c  
gl  
b  
pr

a = 1  
b = 2  
caja  
prin

que b se trate como local,

al.

```
def cajanegra():
```

```
    c = 3
```

```
    global b
```

```
    b = 5
```

```
    print('Dentro de la función b vale', b)
```

```
a = 1
```

```
b = 2
```

```
cajanegra()
```

```
print('Fuera de la función b también vale', b)
```

El resultado de la ejecución sería:

Dentro de la función b vale 5

después de esta línea

El resultado de la ejecución sería:

Fuera de la función b también vale  
5

después de esta línea

después de esta línea



de

El res

Dentro a

**El resultado de la ejecución sería:**

Dentro de la función b vale 5



algo prohibido en Python.

hace que b se trate como local,  
el global.

```
def cajanegra():  
    c = 3  
    global b  
    b = 5  
    print('Dentro de la función b vale', b)
```

```
a = 1  
b = 2  
cajanegra()  
print('Fuera de la función b también vale', b)
```

El resultado de la ejecución sería:  
Dentro de la función b vale 5

después de esta línea

El resultado de la ejecución sería:  
Fuera de la función b también vale  
5

después de esta línea

**Valdrá 5 ambas veces**

de una  
variable  
ción. Para  
variable  
eando la

El resultado

El resultado

Fuera de  
5

después de esta línea



**El resultado de la ejecución sería:**

Fuera de la función b también vale  
5



**¿Qué  
sucederá  
con los  
módulos**

# IPI 2019 - Unidad TRES

UNNOBA



### Parámetros

Los parámetros tienen la misión de comunicar a la función con quien la invoca.  
A los valores que se pasan al invocar a la función se los denomina "argumentos".

```
1. def suma_uno(valor):  
    # Le suma 1 al argumento y lo retorna  
    return valor + 1 # Resultado de la función  
  
# Invocamos la función que definimos:  
2. >>> valor = suma_uno(5)
```

En la línea 1 "valor" es el **parámetro formal** y en la línea 2 el valor 5 es el **parámetro real o argumento**.

### Funciones

#### ¿Cómo modularizar en Python?

En Python contamos con **módulos** y **funciones**.

Comenzamos por las **funciones** sabiendo que contamos con las que provee el propio lenguaje y las que **definimos nosotros**.

Para **definir** una función en Python usamos el siguiente esquema:

```
def nombre_funcion(argumentos):  
    """ Documentación que describe que hace la función """  
    # cuerpo de la función  
    return valor_de_retorno
```

El nombre de la función debe ser en minúsculas y con guiones bajos.

La documentación es una cadena de texto que describe que hace la función.

El cuerpo de la función es el código que se ejecuta cuando se llama a la función.

El valor de retorno es el valor que devuelve la función.

## Modularización

Hasta ahora los problemas vistos son sencillos. Pero en el mundo real los **problemas son más complejos** y de **mayor magnitud**.

Uno de los métodos más conocidos para resolver un problema complejo y grande es **dividirlo** en problemas más pequeños, llamados "subproblemas".

De esta manera, en lugar de resolver una tarea compleja y laboriosa, resolvemos otras más sencillas para que, a partir de ellas, se pueda llegar a la solución buscada originalmente.



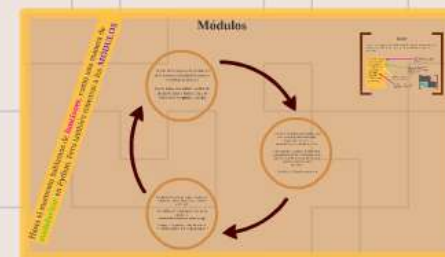
### Ámbito y alcance

El ámbito de una variable se define como el lugar en donde esa variable **sólo** puede ser utilizada.

Las **Locales** son creadas en las funciones. Mueven por las que están fuera de donde fueron creadas. **Globales**.

Las **variables locales** tienen una vida efímera, sólo existen durante el momento en el que se llama la función.

```
x = 10  
  
def suma(x, y):  
    # Variables locales  
    z = x + y  
    return z  
  
# Variables globales  
print(x)  
print(y)  
print(z)  
print(suma(5, 5))
```





Esta presentación fue diseñada por el siguiente equipo docente:

Mg. Claudia Russo  
Lic. Paula Lencina  
AC María Lanzillotta  
Lic. Leticia Galante  
Prog. Trinidad Picco  
AC. Patricia Miguel  
AC M. del Carmen Muller  
Lic. Mariana Adó  
Li. Cecilia Rastelli



**Atribución – No Comercial – Compartir Igual (by-nc-sa):**

No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original. Esta licencia no es una licencia libre.



**Atribución (Attribution):** En cualquier explotación de la obra autorizada por la licencia será necesario reconocer la autoría (obligatoria en todos los casos).



**No Comercial (Non commercial):** La explotación de la obra queda limitada a usos no comerciales.



**Compartir Igual (Share alike):** La explotación autorizada incluye la creación de obras derivadas siempre que mantengan la misma licencia al ser divulgadas.

Por lo que las autoras agradecen el reconocimiento de la autoría correspondiente.

Para mayor información se recomienda acceder a :

<https://creativecommons.org/licenses/by-nc-sa/4.0/>