

PYTHON

UNI – FIIS  
Glen Rodríguez R.

# Listas de 2D

- Elemento se accesa como
  - `<nombre lista>[fila][col]`

`<list_name> = [ [<value 1>, <value 2>, ... <value n>],`  
                  `[<value 1>, <value 2>, ... <value n>],`  
                  `:`                  `:`                  `:`  
                  `:`                  `:`                  `:`  
                  `[<value 1>, <value 2>, ... <value n>] ]` .

The diagram illustrates the structure of a 2D list. A red curly brace on the right side groups the first three rows of the list, labeled "filas" (rows) in red. Another red curly brace at the bottom groups the elements within a single row, labeled "Columnas" (columns) in red. The list is enclosed in square brackets, with a period at the end.

**filas**

**Columnas**

# Ejemplo

```
matriz = [ [0, 0, 0],  
            [1, 1, 1],  
            [2, 2, 2],  
            [3, 3, 3]]
```

```
for r in range (0, 4, 1):
```

```
    print (matriz[r]) # cada print muestra una lista 1D
```

```
for r in range (0,4, 1):
```

```
    for c in range (0,3,1):
```

```
        print(matriz[r][c], end="")
```

```
    print()
```

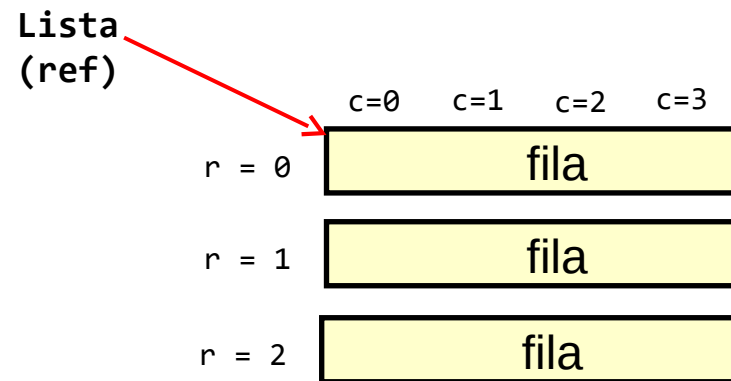
```
print(matriz[2][0]) #2 no 0
```

```
r = 0 [0, 0, 0]  
r = 1 [1, 1, 1]  
r = 2 [2, 2, 2]  
r = 3 [3, 3, 3]
```

```
      012 (col)  
r = 0 000  
r = 1 111  
r = 2 222  
r = 3 333
```

# Como crearlas en ejecución

- Crear una lista 1D (referencia)
- Hacer bucle a lo largo de esa lista 1D
- En cada iteración crear otra lista 1D
- Hacer un bucle interno que cree los elementos de esa segunda lista
- Repetir el proceso



# Ejemplo

```
aGrid = []                # Crea referencia a lista
for r in range (0, 3, 1): # Bucle externo para cada fila
    aGrid.append ([])      # Crear fila (lista 1D)
    for c in range (0, 3, 1): # Bucle interno para cada columna
        aGrid[r].append (" ") # Crear los elementos
```

# Ejemplo

```
aGrid = []
```

```
for r in range (0,2,1):  
    aGrid.append ([])  
    for c in range (0,3,1):  
        aGrid[r].append (str(r+c))
```

```
for r in range (0,2,1):  
    for c in range (0,3,1):  
        print(matrix[r][c], end="")  
    print()
```

# Ojo

- Las listas de Python pueden ser heterogéneas (elementos de diferentes tipos)
- Ejemplo
- `aList = ["pepe", "ana", 15, 7.5, True]`
- `BList= [ [12, "hola"] , ["abc", 3.4] ]`

# Tuplas

- Parecido a una lista
- Pero son inmutables
- Se usan para guardar data que NO debería cambiar



# Creando tuplas

## Formato:

*nombre\_tupla = (valor<sup>1</sup>, valor<sup>2</sup>...valor<sup>n</sup>)*

## Ejemplo:

tup = (1,2,"UNI",0.3)

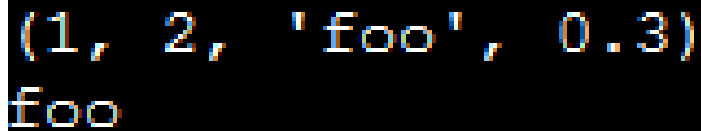
# Ejemplo

```
tup = (1,2,"foo",0.3)
```

```
print (tup)
```

```
print (tup[2])
```

```
tup[2] = "bar"
```



```
(1, 2, 'foo', 0.3)  
foo
```

**Error (tratar de cambiar un immutable):**

← "TypeError: object does not support item assignment"

# Tupla vacía y de 1 elemento

- $t1 = ()$
- $t2 = (34,)$
- Una tupla no se puede cambiar pero si eliminar del todo:
- del  $t2$
- Y ya no existe  $t2$ .

# Del y Remove

- En listas puede borrar un solo elemento

```
values = [100, 200, 300, 400, 500, 600]
```

```
# Use del tpara remover 1 o varios indice.
```

```
del values[2]
```

```
print(values)
```

```
Values = [100, 200, 300, 400, 500, 600, 500]
```

```
# Use remove para borrar 1 solo valor.
```

```
values.remove(500)
```

```
print(values)
```

# Del y remove

```
elements = ["A", "B", "C", "D"]  
# Remove los dos de en medio.  
del elements[1:3]  
print(elements)
```

# Operando en tuplas

```
tup1 = (12, 34.56)
```

```
tup2 = ('abc', 'xyz')
```

```
tup3 = tup1 + tup2;
```

```
print tup3
```

```
tup4 = 3*tup2;
```

```
print tup4
```

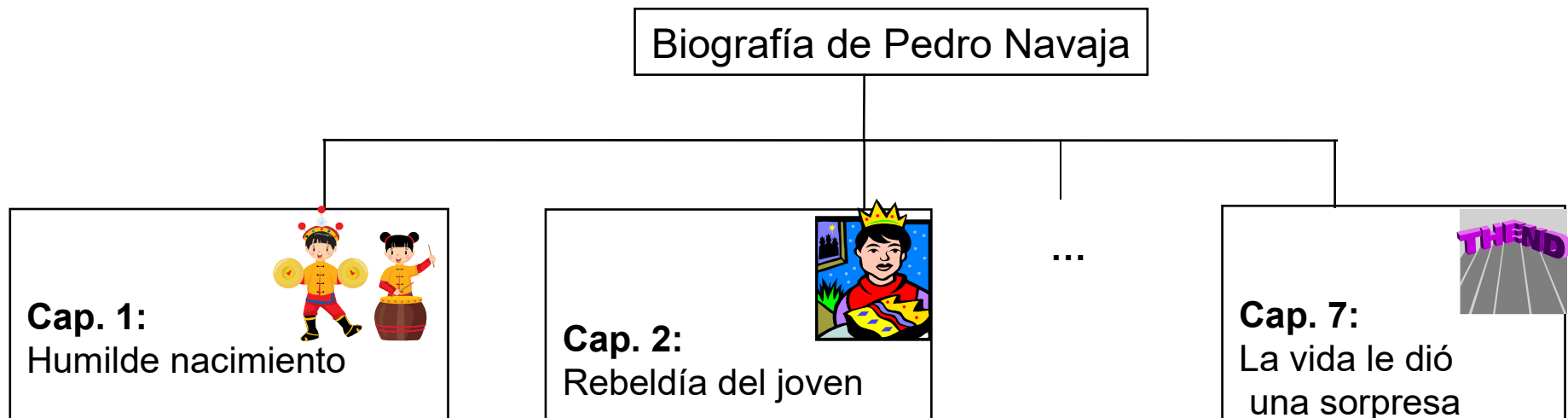
```
lista= [4 , 6 , 8, 10]
```

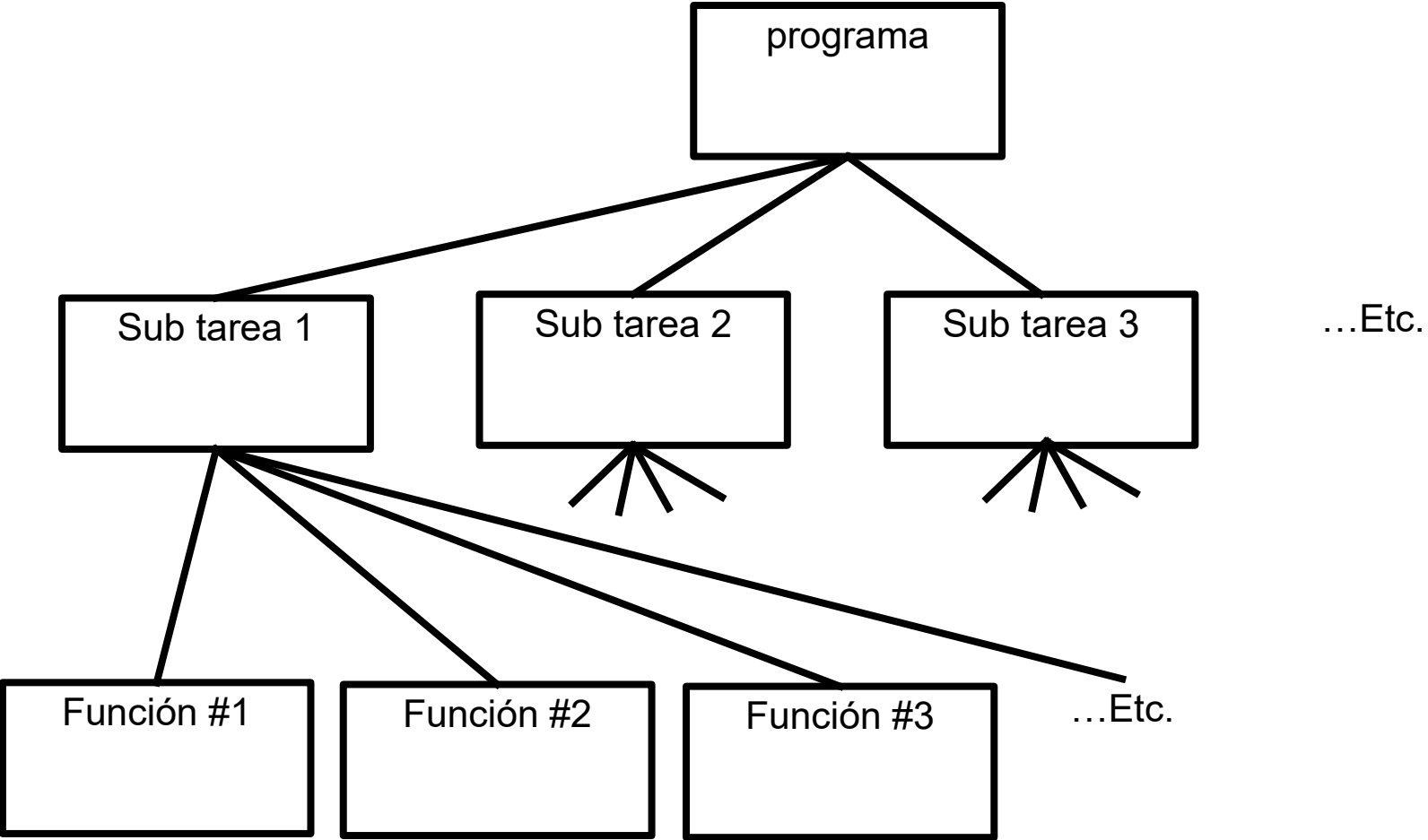
```
tup5=tuple(lista)
```

```
print(tup5)
```

# Funciones

- Principio de descomponer un problema en sub-problemas para hacerlo más manejable
- Enfoque top-down







# Ejemplo

- Programa que debe notificar la mora de un deudor de tarjeta de crédito
  - Leer los datos del deudor y la deuda
  - Calcular la mora
  - Imprimir el documento de cobranza

# Definiendo una función

## Formato:

```
def <nombre de la función>():  
    cuerpo1
```

## Ejemplo:

```
def displayInstructions():  
    print ("Aca se ven las instrucciones")
```

# Llamando a la función

## Formato:

`<function name>()`

## Ejemplo:

`displayInstructions()`

# Como se diferencia las funciones del programa principal

- Las funciones tienen el “def” y el cuerpo está indentado
- El programa principal no esta indentado. Toda línea pegada al margen izquierdo es parte del programa principal.
- Se recomienda que las funciones se definan antes de ser usadas

# Ejemplo

```
def displayInstructions():
```

```
    print("Displaying instructions")
```

```
    Displaying instructions
```

```
# Aca empieza el programa principal, no hay indentación
```

```
displayInstructions()
```

```
print("End of program")
```

```
End of program
```

# Ejemplo

```
def displayInstructions():  
    print("Displaying instructions")
```

**Definición de  
la función**



**# Programa principal**

```
displayInstructions()  
print("End of program")
```

**Llamada a la  
función**



# Haciendo del prog.principal otra función

```
def displayInstructions():  
    print ("Displaying instructions")
```

```
def start():  
    displayInstructions()  
    print("End of program")
```

```
start()
```

```
# Algunos prefieren llamarla main en vez de start
```

# Alcance de las variables

- Variables locales: creadas dentro del cuerpo de una función (indentadas)
- Globales: creadas fuera de la función. Constantes.

```
HUMAN_CAT_AGE_RATIO = 2
```

```
def getInformation():
```

```
    age = input("What is your age in years: ")
```

```
    catAge = age * HUMAN_CAT_AGE_RATIO
```

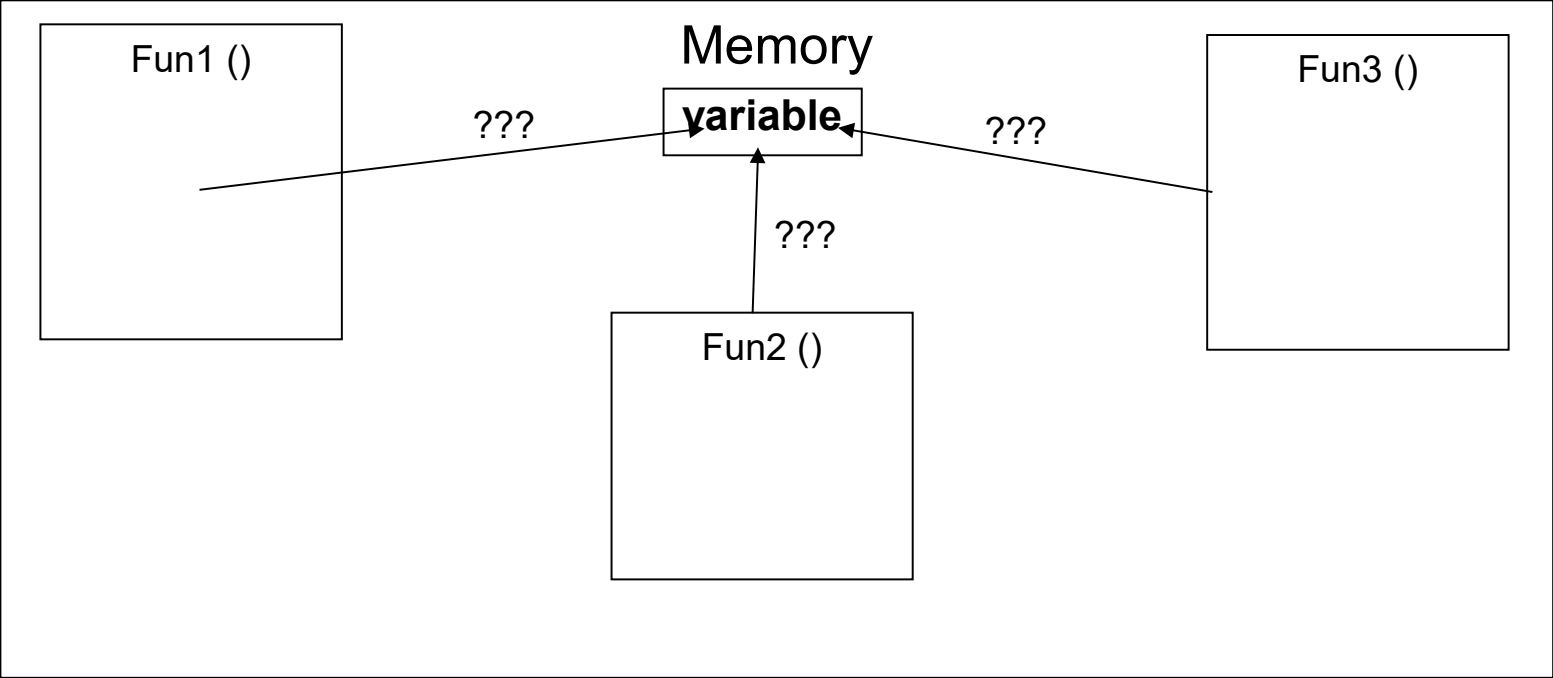
Global

Local



# Las variables dentro de la función

- Se crean (se les asigna memoria) cuando se llama a la función
- Al terminar la función, se destruyen (la memoria se libera)
- Por lo tanto, cualquier cambio que sufran no afecta nada fuera de la función
- Las globales no deben cambiarse (efectos espurios) → constantes



```
def fun():  
    x = 7  
    y = 13
```

Memory: 'fun'

x 7

y 13

**Impossible (bien!)**

```
def start():  
    a = 1  
    fun()  
    # x,y  
    # inaccessible
```

Memory:

'start'

a 1

# Cómo acceder a los valores de función a principal y viceversa?

- Opción 1: paso de parámetros

## Formato:

```
def <nombre función>(<parámetro 1>, <parámetro 2>...  
    <parámetro n-1>, <parámetro n>):
```

## Ejemplo:

```
def display(celsius, fahrenheit):
```

## Ejemplo de llamada:

```
display(34, var2)
```

# Nota

- Los parámetros son variables de la función

```
def fun(num1):  
    print(num1)  
  
    num2 = 20  
  
    print(num2)
```

```
def start():  
    num1 = 1  
  
    fun(num1)
```

```
start()
```

# Ejemplo

```
def introduction
    print("Bienvenidos a este programa")

def display (celsius, fahrenheit):
    print ("")
    print ("Celsius value: ", celsius)
    print ("Fahrenheit value:", fahrenheit)

def convert ():
    celsius = float(input ("Type in the celsius temperature: "))
    fahrenheit = celsius * 9 / 5 + 32
    display (celsius, fahrenheit)

# start function

def start ():
    introduction ()
    convert ()

start ()
```

# Errores típicos

- La función tiene diferente número de parámetros en su definición y en una llamada
  - `display(45)`
  - `display(45, 5, 6)`
- Tipos no coinciden
  - `display(45,"hola")`
- Variable parámetro no creada
  - `display(45,var2) #aún no haz creado var2`

# Alcances

- Son las líneas de código donde se puede usar una variable o constante
- Al ser declarado empieza el alcance
- Hasta que bloque donde fue declarado
- Osea, una variable dentro de un bucle sólo “existe” en ese bucle, no afuera
- Lo mismo con una función



# Ejemplo

- Qué pasa con “num”?

```
def fun1():  
    num = 10  
  
    # ...  
  
    # ...  
  
    # Fin de fun1
```

```
def fun2():  
    fun1()  
  
    num = 20
```

# Y cómo pasa los cálculos hechos en la función afuera?

Var.local  
interest = 50

```
def calculateInterest(principle, rate, time):  
    interest = principle * rate * time
```

# start

```
principle = 100
```

```
rate = 0.1
```

```
time = 5
```

```
calculateInterest (principle, rate, time)
```

```
print("Interest earned $", interest)
```

**Problema:**

Var.local de la  
función ya no  
existe fuera de la  
función

# Solución: return

```
def calculateInterest(principle, rate, time):
```

```
    interest = principle * rate * time
```

```
    return(interest)
```

**Variable  
'interest' sigue  
siendo local**

```
# start
```

```
    principle = 100
```

```
    rate = 0.1
```

```
    time = 5
```

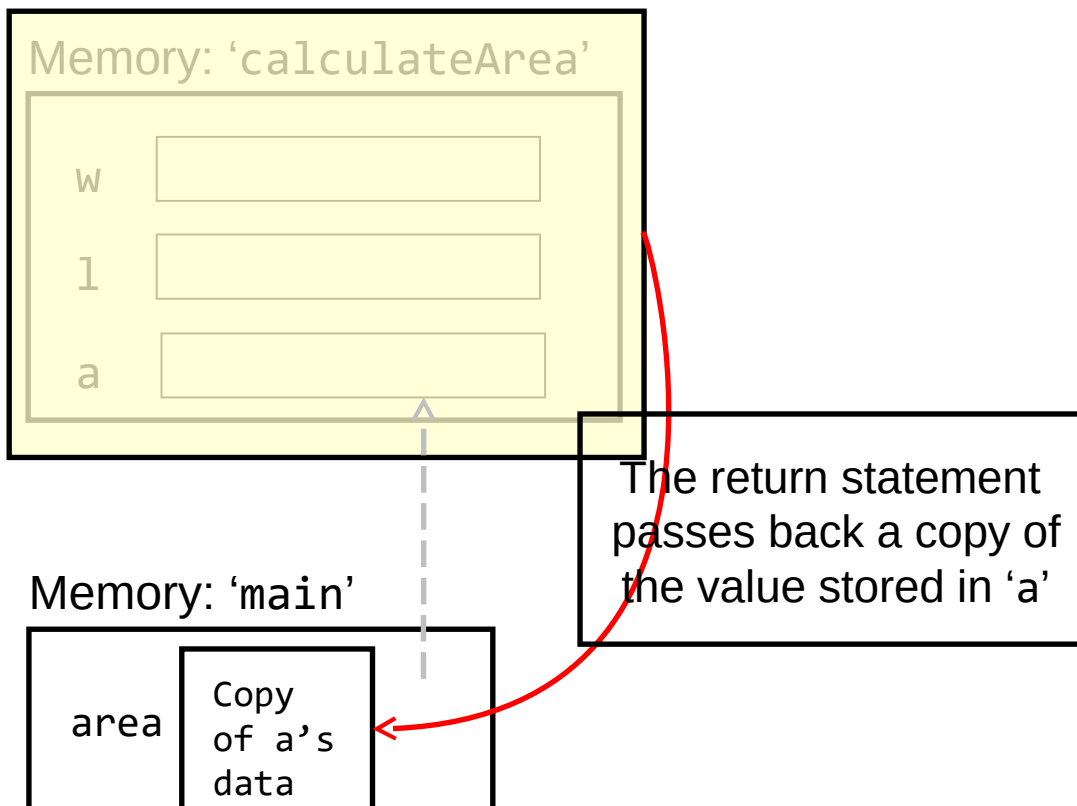
```
    interest = calculateInterest(principle,  
                                rate, time)  
    print ("Interest earned $", interest)
```

**El valor se copia en otra  
variable que si se puede  
usar fuerade la función**

## RAM

```
def calculateArea():  
    w = int(input())  
    l = int(input())  
    a = w * l  
    return(a)
```

```
def main():  
    area = calculateArea()  
    print(area)
```



# Return

## Formato (un solo valor):

<code>return(&lt;valor&gt;)</code>	<code># Definicion de funcion</code>
<code>&lt;variable&gt; = &lt;funcione&gt;()</code>	<code># Llamada a funcion</code>

## Ejemplo:

<code>return(interest)</code>	<code># Definicion</code>
<code>interest = calculateInterest</code> <code>(principle, rate, time)</code>	<code># Llamada</code>

# Return

## Formato (Multiples valores):

# definicion

```
return(<valor1>, <valor2>...)
```

# llamada

```
<variable 1>, <variable 2>... = <funcion>()
```

## Ejemplo (Multiples valores):

# definicion

```
return(principle, rate, time)
```

# llamada

```
principle, rate, time = getInputs(filename, number)
```

# Ejemplo mejorado

```
def getInputs():  
    principle = float(input("Enter the original principle: "))  
    rate = float(input("Enter the yearly interest rate %"))  
    rate = rate / 100  
    time = input("Enter the number of years that money will be invested:  
                ")  
    time = float(time)  
    return(principle, rate, time)  
  
def calculate(principle, rate, time):  
    interest = principle * rate * time  
    amount = principle + interest  
    return(interest, amount)
```

# Ejemplo mejorado

```
def display(principle, rate, time, interest, amount):  
    temp = rate * 100  
    print("")  
    print("Investing $%.2f" %principle, "at a rate of %.2f" %temp, "%")  
    print("Over a period of %.0f" %time, "years...")  
    print("Interest accrued $", interest)  
    print("Amount in your account $", amount)
```



# Ejemplo mejorado

```
def start():  
    principle = 0  
    rate = 0  
    time = 0  
    interest = 0  
    amount = 0  
  
    introduction ()  
    principle, rate, time = getInputs()  
    interest, amount = calculate(principle, rate, time)  
    display(principle, rate, time, interest, amount)  
  
start()
```

# Otro error frecuente

- Olvidarse de la variable donde se guarda el valor retornado
- En vez de `d= distancia(a,b)`
- `distancia(a,b)`
- Pero esta permitido funciones sin return. Allí si se debe omitir esa variable

# Principio de caja negra

- Aunque una variable del programa principal pudiese ser leída y modificada en una función, NO SE DEBERIA!
- La función debería usar SOLO los parámetros que le pasan. El resto de variable que use son locales.
- Si toco variables “afuera”, podría “sin querer” alterar otra parte del programa
- Python lo soluciona: no te deja, a menos que usen “global”!

# Ejemplo

```
num = 1
```

```
def fun():  
    global num  
    num = 2  
    print(num)
```

```
def start():  
    print(num)  
    fun()  
    print(num)
```

```
start()
```

# Definir antes de llamar !

- **Correcto** 😊

```
def fun():  
    print("Bien")
```

} Def.

# start  
fun() }

llamada

- **Incorrecto** ☹️

# Start  
fun() }

llamada

```
def fun():  
    print("Mal")
```

} Def.

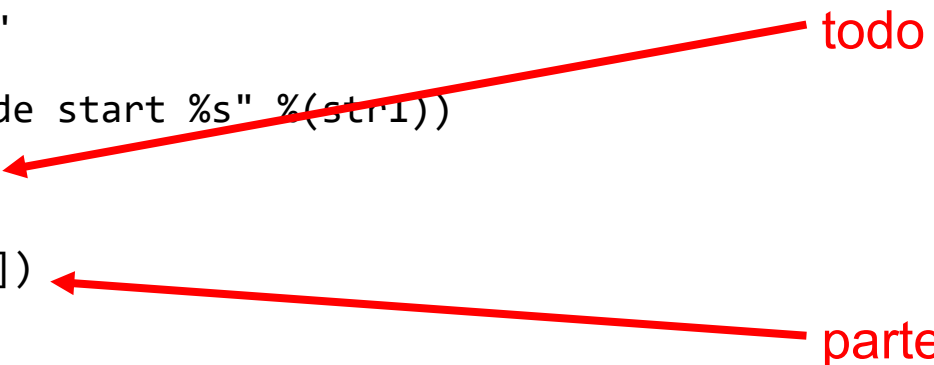
# Strings en funciones

- Strings como parámetros (todo o parte)

```
def fun1(str1):  
    print("Inside fun1 %s" %(str1))
```

```
def fun2(str2):  
    print("Inside fun2 %s" %(str2))
```

```
def start():  
    str1 = "abc"  
    print("Inside start %s" %(str1))  
    fun1(str1)  
    fun2(str1[1])
```



todo

parte

# Listas en funciones

- Si paso una lista como parámetro, paso la copia de LA DIRECCION
- O sea, si modifico la “copia” en la función, modifico la lista original
- Por eso, mucho cuidado con el uso de listas en funciones!

# Ejemplo

```
def fun(aReference):  
    aReference[1] = 8  # NO !
```

```
def start():  
    aReference = [1,2,3]  
    fun(aReference)  
    print(aReference)
```



# Funciones recursivas

- Se denomina recursivas a aquellas funciones que en su algoritmo, hacen referencia sí misma.
- Las funciones recursivas suelen ser muy útiles en casos muy puntuales, pero debido a su gran factibilidad de caer en iteraciones infinitas, deben extremarse las medidas preventivas adecuadas y, solo utilizarse cuando sea estrictamente necesario y no exista una forma alternativa viable, que resuelva el problema evitando la recursividad.

# Ejemplo

```
def fib(n):  
    if n <= 1:  
        return n  
    t = fib(n-1) + fib(n-2)  
    return t
```

# Ejemplo

```
from turtle import *
```

```
def koch(a, order):
```

```
    if order > 0:
```

```
        for t in [60, -120, 60, 0]:
```

```
            koch(a/3, order-1)
```

```
            left(t)
```

```
    else:
```

```
        forward(a)
```

```
koch(150,2)
```

# Ojo

- La recursión puede ser muy lenta
- Debe haber una condición de parada, por ejemplo, factorial de 1 es 1.
- La llamada recursiva debe ser a un caso más simple, no más complicado, que el original. Factorial de 3 llama al factorial de 2 (que es más simple)

# Modulos

- Si creo un archivo .py con una o más funciones y lo grabo en disco, puedo usar esas funciones en un programa diferente
- Suponga que creo dos funciones sumar(a,b) y restar(a,b) en un archivo llamado misfunciones.py
- Desde otro programa hago:

```
import misfunciones
```

```
x=misfunciones.sumar(2,6)
```

# Modulos

- Si el nombre del archivo es muy grande, se puede abreviar

```
mfr=misfunciones.restar
```

```
y=mfr(6,4)
```

- Las variables del modulo son invisibles e intocables para el programa que las importa

# Conjuntos (set)

- Como en matemáticas: grupo de elementos SIN repeticiones y donde el orden no importa (no son secuencias)
- Tiene operaciones de union, intersección, etc., como conjuntos matemáticos

# Ejemplo

```
basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
```

```
fruit = set(basket)          # crear un set
```

```
print(fruit)
```

```
print('orange' in fruit)     # pertenece a?
```

```
print('crabgrass' in fruit)
```

```
a = set('abracadabra')
```

```
b = set('alacazam')
```

```
print(a)
```

```
print(b)
```

```
print(a-b)
```

```
print(a | b)
```

```
print(a & b)
```

```
print(a ^ b)
```



# Diccionario

- Colección no-ordenada de valores que son accedidos a través de una clave. Es decir, en lugar de acceder a la información mediante el índice numérico, como es el caso de las listas y tuplas, es posible acceder a los valores a través de sus claves, que pueden ser de diversos tipo.
- En otros lenguajes: hash

# Diccionario

- Las claves son únicas dentro de un diccionario, es decir que no puede haber un diccionario que tenga dos veces la misma clave, si se asigna un valor a una clave ya existente, se reemplaza el valor anterior.
- No hay una forma directa de acceder a una clave a través de su valor, y nada impide que un mismo valor se encuentre asignado a distintas claves
- La información almacenada en los diccionarios, no tiene un orden particular. Ni por clave ni por valor, ni tampoco por el orden en que han sido agregados al diccionario.

# Ejemplo

```
punto = {'x': 2, 'y': 1, 'z': 4}
```

```
print(punto["z"])
```

```
materias = {}
```

```
materias["lunes"] = [6103, 7540]
```

```
materias["martes"] = [6201]
```

```
materias["miércoles"] = [6103, 7540]
```

```
materias["jueves"] = []
```

```
materias["viernes"] = [6201]
```

```
print materias["lunes"]
```

# Ejemplos

```
for dia in materias:
```

```
    print (dia, ":", materias[dia])
```

```
print materias["domingo"]
```

```
print materias.get("domingo")
```

# Ver si hay una clave

```
d = {'x': 12, 'y': 7}
```

```
if d.has_key('x'):
```

```
    print d['x']  # Imprime 12
```

```
if d.has_key('z'):
```

```
    print d['z']  # No se ejecuta
```

```
if 'y' in d:
```

```
    print d['y']  # Imprime 7
```

# Acelerando recursión con diccionarios

```
def fib(n):  
    if n in fib.cache:  
        return fib.cache[n]  
    ret = fib(n-2) + fib(n-1)  
    fib.cache[n] = ret  
    return ret  
  
fib.cache = {0: 1, 1: 1}
```

# Colecciones

- Tipos de data nuevos en Python. Hay varios
- Count: para convertir una lista simple en un diccionario de frecuencias de los elementos. Ej: ['a', 'b', 'c', 'a', 'b', 'b'] en {'b': 3, 'a': 2, 'c': 1}
- Deque: para implementar colas y pilas.
- OrderedDict: diccionario con orden.