

Fundamentos de Algoritmos

Prof. Agustín Gravano
Prof. Hernán Czemerinski

Clase 2: Estructuras de control y funciones

Programa

Un **programa** es una secuencia finita de **instrucciones**.

Ejemplo:

Ingredientes: 15 huevos, 600 gramos de harina, 600 gramos de azucar

- 1.- Mientras no estén espumosos, batir los huevos junto con el azúcar,
- 2.- agregar la harina en forma envolvente sin batir,
- 3.- batir suavemente,
- 4.- colocar en el horno a 180 grados,
- 5.- si le clavo un cuchillo y sale húmedo, entonces ir a 4.-
- 6.- retirar del horno,
- 7.- mientras no esté frío, esperar
- 8.- desmoldar y servir

Instrucción

Una **instrucción** es una operación que:

- transforma los datos (el *estado*), o bien
- modifica el flujo de ejecución.

- 1.- Mientras no estén espumosos, batir los huevos junto con el azúcar,
- 2.- agregar la harina en forma envolvente sin batir,
- 3.- batir suavemente,
- 4.- colocar en el horno a 180 grados,
- 5.- si le clavo un cuchillo y sale húmedo, entonces ir a 4.-
- 6.- retirar del horno,
- 7.- mientras no esté frío, esperar
- 8.- desmoldar y servir

Variable y Asignación

Una **variable** es un **nombre** que denota la **dirección** de una celda en la memoria, en la cual se almacena un **valor**.

En esa celda de memoria es posible:

- **leer** el valor almacenado, y
- **escribir** un valor nuevo, que reemplace al anterior.

VARIABLE = EXPRESIÓN

Almacena el valor de la *EXPRESIÓN* en la dirección en memoria denotada por *VARIABLE*.

Ejemplos: **x = 1000**
 x = x + y * 22

Estado

Se denomina **estado** al valor de todas las variables de un programa en un punto de su ejecución.

Es una “foto” de la memoria en un momento determinado.

Ejemplo:

x:?	y:?
-----	-----

y = 10

x:?	y:10
-----	------

x = y * 2

x:20	y:10
------	------

y = y + 1

x:20	y:11
------	------

Estructuras de control de flujo

- Condicionales
- Ciclos
- Funciones



Condicionales



Piedra, papel o tijera

Reglas

- Si un jugador elige piedra y el otro papel, entonces gana el que elige papel
- Si un jugador elige papel y el otro tijera, entonces gana el que elige tijera

...

Si *CONDICIÓN*, entonces *ALGO*

Condicionales

if *CONDICIÓN1:*

BLOQUE1

CONDICIÓN1 y 2 son expresiones booleanas.

BLOQUE1, 2 y 3 son bloques de código.

elif *CONDICIÓN2:*

BLOQUE2

BLOQUE1 se ejecuta **sii** *CONDICIÓN1* es verdadera.

BLOQUE2, **sii** *CONDICIÓN1* es falsa y *CONDICIÓN2* es verdadera.

else:

BLOQUE3

BLOQUE3, **sii** *CONDICIÓN1* y *CONDICIÓN2* son falsas.

Ejemplo:

```
if x > 5:
    print("x es mayor que 5.")
elif x == 5:
    print("x es igual a 5.")
else:
    print("x es menor que 5.")
```

Podemos poner cualquier número de elif's.

Condicionales

Ejercicio:

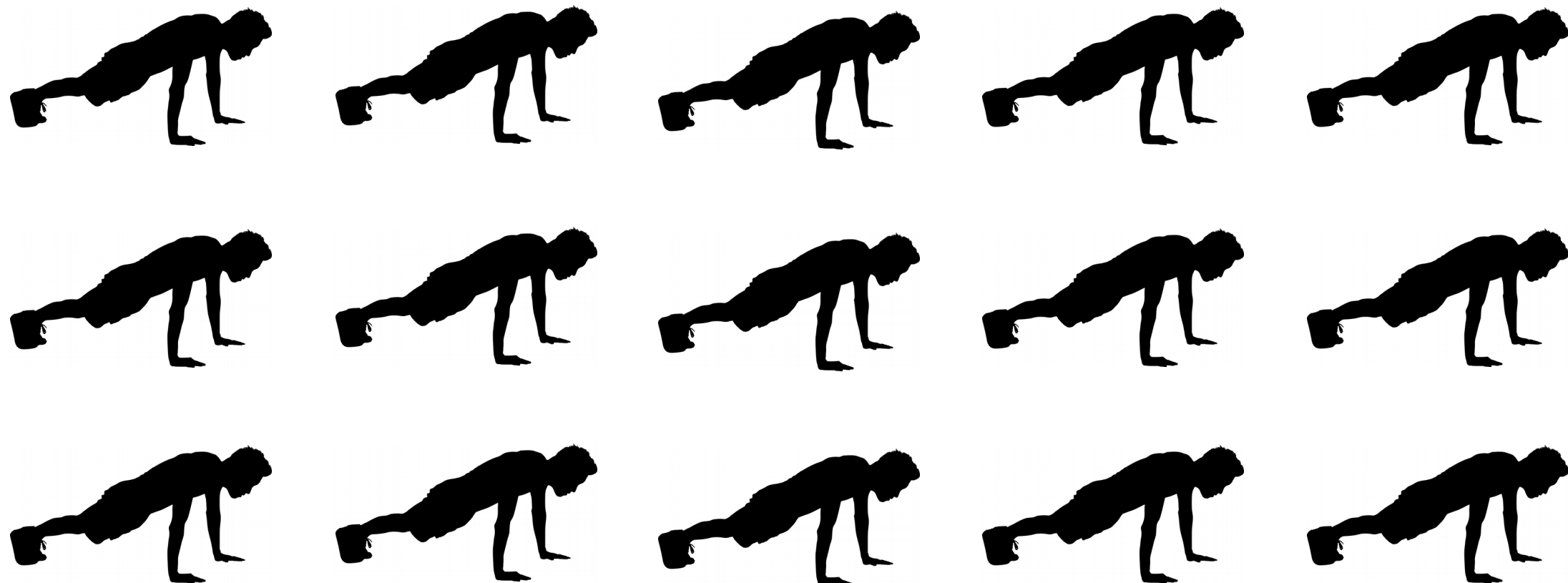
¿Qué imprime por pantalla este código?

```
a = 1234
b = 987
if a > b:
    c = a
else:
    c = b
print(c)
```

La indentación
de los bloques de código
es OBLIGATORIA en Python.

Primero pensar qué imprime y luego ejecutarlo en Spyder y observar lo que sucede.

Ciclos



Repetir 15



Ciclos

while *CONDICIÓN*:
BLOQUE

CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
→ i = 0
while i < 3:
    print(i)
    i = i + 1
```

Memoria	
i	?

Consola	

Ciclos

while *CONDICIÓN*:
BLOQUE

CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
i = 0  
→ while i < 3:  
    print(i)  
    i = i + 1
```

Memoria	
i	0

Consola	

Ciclos

while *CONDICIÓN*:
BLOQUE

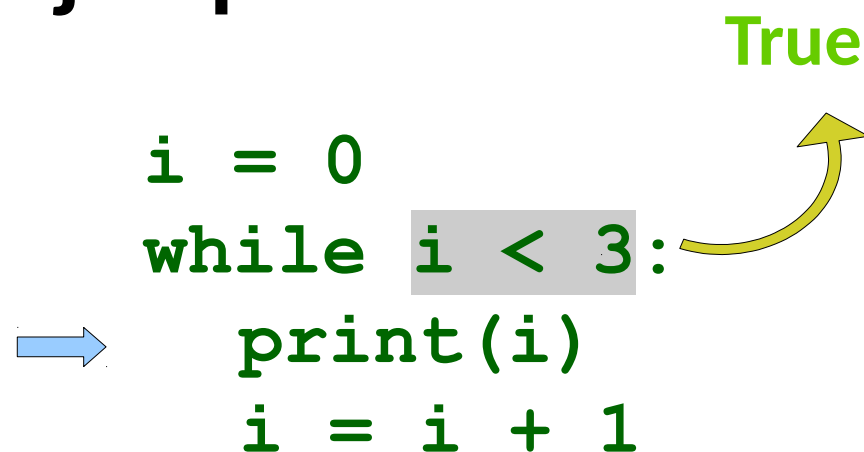
CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
i = 0
while i < 3:
    print(i)
    i = i + 1
```

True



Memoria	
i	0

Consola

Ciclos


while *CONDICIÓN*:
BLOQUE

CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
i = 0
while i < 3:
    print(i)
    i = i + 1
```



Memoria	
i	0

Consola	
0	

Ciclos

while *CONDICIÓN*:
BLOQUE

CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
i = 0  
→ while i < 3:  
    print(i)  
    i = i + 1
```

Memoria	
i	1

Consola	
0	

Ciclos

while *CONDICIÓN*:
BLOQUE

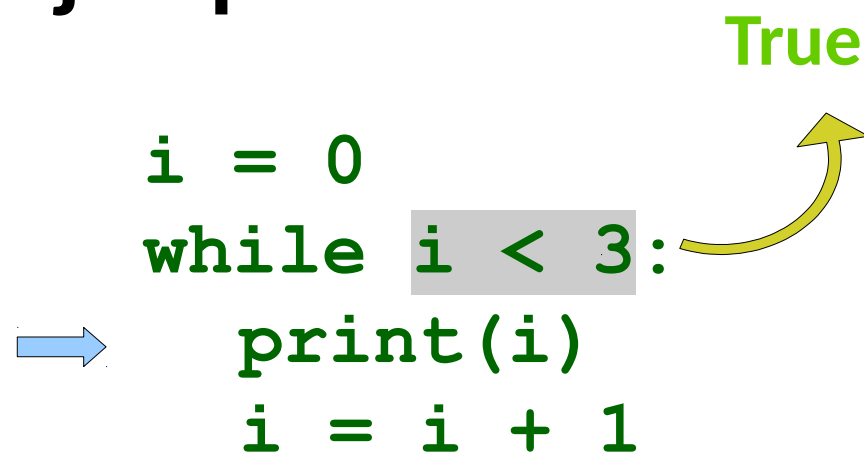
CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
i = 0
while i < 3:
    print(i)
    i = i + 1
```

True



Memoria	
i	1

Consola
0

Ciclos


while *CONDICIÓN*:
BLOQUE

CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
i = 0
while i < 3:
    print(i)
    i = i + 1
```



Memoria	
i	1

Consola	
0	
1	

Ciclos

while *CONDICIÓN*:
BLOQUE

CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
i = 0  
→ while i < 3:  
    print(i)  
    i = i + 1
```

Memoria	
i	2

Consola	
0	
1	

Ciclos

while *CONDICIÓN*:
BLOQUE

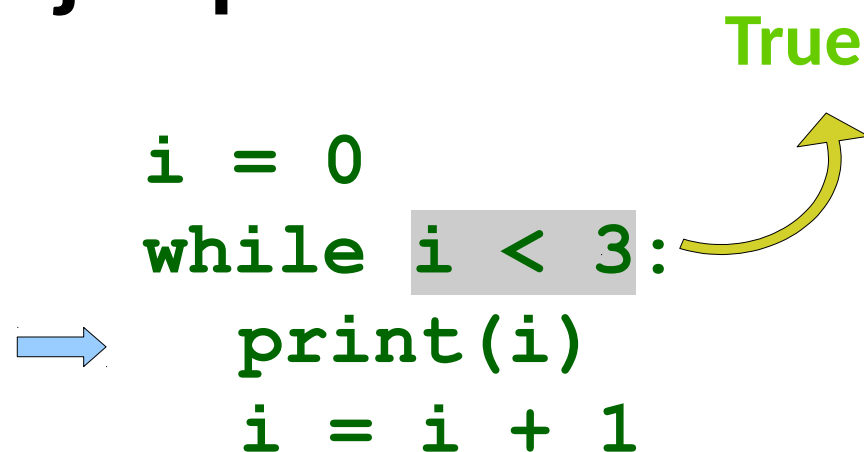
CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
i = 0
while i < 3:
    print(i)
    i = i + 1
```

True



Memoria	
i	2

Consola	
0	
1	

Ciclos


while *CONDICIÓN*:
BLOQUE

CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
i = 0
while i < 3:
    print(i)
    i = i + 1
```



Memoria	
i	2

Consola	
0	
1	
2	

Ciclos

while *CONDICIÓN*:
BLOQUE

CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
i = 0  
→ while i < 3:  
    print(i)  
    i = i + 1
```

Memoria	
i	3

Consola	
0	
1	
2	

Ciclos

while *CONDICIÓN*:
BLOQUE

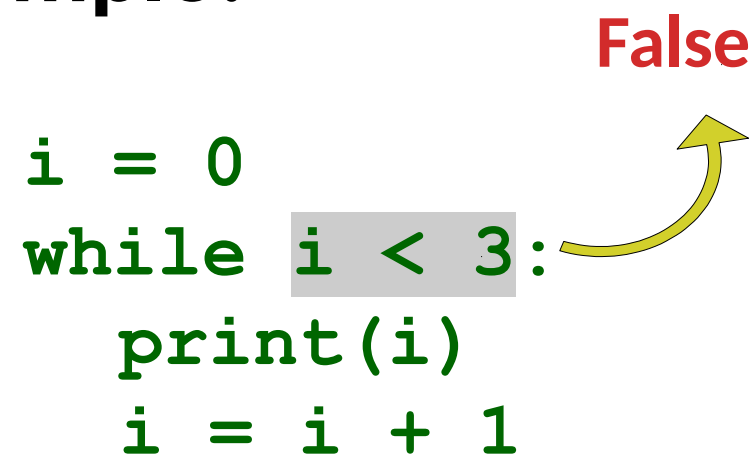
CONDICIÓN es una expresión booleana.
BLOQUE es un bloque de código.

BLOQUE se ejecuta una y otra vez,
mientras *CONDICIÓN* siga siendo **verdadera**.

Ejemplo:

```
i = 0
while i < 3:
    print(i)
    i = i + 1
```

False



Memoria	
i	?

Consola	
0	
1	
2	

Ciclos

Ejercicio 1:

Observar la diferencia que se produce al ejecutar los siguientes dos programas en Python:

Programa 1

```
print(1)
print(2)
```

Programa 2

```
print(1, end=' ')
print(2, end=' ')
```

Ejercicio 2:

Escribir un programa en Python que, usando un ciclo imprima por pantalla los números del 1 al 5 en una única línea.

Ejercicio 3:

Escribir un programa en Python que usando ciclos imprima por pantalla lo que muestra la imagen.

```
Ana: ¡Yo tengo escalera!
1 2 3 4 5
Agustín: ¡Yo también!
1 2 3 4 5
```

Funciones

De la Sierra Morena,
cielito lindo, vienen bajando
un par de ojitos negros,
cielito lindo, de contrabando.

¡Ay, ay, ay, ay!
canta y no llores,
porque cantando se alegran,
cielito lindo, los corazones.

Ese lunar que tienes,
cielito lindo, junto a la boca
no se lo des a nadie,
cielito lindo, que a mí me toca.

¡Ay, ay, ay, ay!
canta y no llores,
porque cantando se alegran,
cielito lindo, los corazones.

Siempre que te enamores
mira primero, mira primero
dónde pones los ojos,
cielito lindo, no llores luego.

¡Ay, ay, ay, ay!
canta y no llores,
porque cantando se alegran,
cielito lindo, los corazones.

De tu reja a la mía,
cielito lindo, no hay más que un paso
ora que estamos solos,
cielito lindo, dame un abrazo.

¡Ay, ay, ay, ay!
canta y no llores,
porque cantando se alegran,
cielito lindo, los corazones.



Funciones

De la Sierra Morena,
cielito lindo, vienen bajando
un par de ojitos negros,
cielito lindo, de contrabando.

[Estribillo]

Ese lunar que tienes,
cielito lindo, junto a la boca
no se lo des a nadie,
cielito lindo, que a mí me toca.

[Estribillo]

Siempre que te enamores
mira primero, mira primero
dónde pones los ojos,
cielito lindo, no llores luego.

[Estribillo]

De tu reja a la mía,
cielito lindo, no hay más que un paso
ora que estamos solos,
cielito lindo, dame un abrazo.

[Estribillo]

Estribillo

¡Ay, ay, ay, ay!
canta y no llores,
porque cantando se alegran,
cielito lindo, los corazones.



Funciones

```
print('Ana: ¡Yo tengo escalera!')
i = 1
while i < 6:
    print(i, end=' ')
    i = i + 1
print()
print('Agustín: ¡Yo también!')
i = 1
while i < 6:
    print(i, end=' ')
    i = i + 1
print()
print('Pablo: ¡Mirá qué casualidad!')
i = 1
while i < 6:
    print(i, end=' ')
    i = i + 1
print()
print('María: ¡Esto es insólito!')
i = 1
while i < 6:
    print(i, end=' ')
    i = i + 1
print()
print('Julietta: ¿Esto sería un empate?')
i = 1
while i < 6:
    print(i, end=' ')
    i = i + 1
print()
print('Olivia: ¡Obvio!')
i = 1
while i < 6:
    print(i, end=' ')
    i = i + 1
```

Funciones

```
i = 1
while i < 6:
    print(i, end=' ')
    i = i + 1
print()
```

- Código con **entidad propia**: imprime los números del 1 al 5
- Resuelve un **subproblema** dentro de un problema más grande
- El mismo código se repite muchas veces en forma idéntica

Funciones

```
print('Ana: ¡Yo tengo escalera!')
imprimir_escalera()
print('Agustín: ¡Yo también!')
imprimir_escalera()
print('Pablo: ¡Mirá qué casualidad!')
imprimir_escalera()
print('María: ¡Esto es insólito!')
imprimir_escalera()
print('Julietta: ¿Esto sería un empate?')
imprimir_escalera()
print('Olivia: ¡Obvio!')
imprimir_escalera()
```

Funciones

Una **función** es una unidad de código que **aísla** una parte de un cómputo. Es un programa dentro de un programa.

- Permite dividir un problema en **problemas más simples**.
- Permite **ordenar** conceptualmente el código para que sea más fácil de entender.
- Permite **reutilizar** soluciones a problemas pequeños en la solución de problemas mayores.

Funciones

def *NOMBRE*:
BLOQUE

NOMBRE es el nombre de la función.

BLOQUE es un bloque de código.

BLOQUE se ejecuta cada vez que la función *NOMBRE* es invocada.

Ejemplo:

La palabra **def** indica que estamos definiendo una función

```
def millas_a_kilometros():  
    mi = 50  
    km = mi * 1.60934  
    return km
```

```
print(50, 'millas equivalen a', millas_a_kilometros(), 'kilómetros')
```

Funciones

def *NOMBRE*:
BLOQUE

NOMBRE es el nombre de la función.
BLOQUE es un bloque de código.

BLOQUE se ejecuta cada vez que
la función *NOMBRE* es invocada.

Ejemplo:

A continuación se escribe el nombre de la
función. ¡Usar nombres descriptivos!

```
def millas_a_kilometros():  
    mi = 50  
    km = mi * 1.60934  
    return km
```

```
print(50, 'millas equivalen a', millas_a_kilometros(), 'kilómetros')
```

Funciones

def *NOMBRE*:
BLOQUE

NOMBRE es el nombre de la función.
BLOQUE es un bloque de código.

BLOQUE se ejecuta cada vez que la función *NOMBRE* es invocada.

Ejemplo:

Los paréntesis se usan para definir parámetros (ya lo veremos); los dos puntos indican que lo que sigue a continuación es el bloque de código que compone la función.

```
def millas_a_kilometros():  
    mi = 50  
    km = mi * 1.60934  
    return km
```

```
print(50, 'millas equivalen a', millas_a_kilometros(), 'kilómetros')
```


Funciones

def *NOMBRE*:
BLOQUE

NOMBRE es el nombre de la función.

BLOQUE es un bloque de código.

BLOQUE se ejecuta cada vez que la función *NOMBRE* es invocada.

Ejemplo:

```
def millas_a_kilometros():  
    mi = 50  
    km = mi * 1.60934  
    return km
```

Bloque de código que compone el cuerpo de la función. Al igual que cualquier bloque, se usan TABs para delimitarlo.

```
print(50, 'millas equivalen a', millas_a_kilometros(), 'kilómetros')
```

Funciones

def *NOMBRE*:
BLOQUE

NOMBRE es el nombre de la función.
BLOQUE es un bloque de código.

BLOQUE se ejecuta cada vez que la función *NOMBRE* es invocada.

Ejemplo:

```
def millas_a_kilometros():  
    mi = 50  
    km = mi * 1.60934  
    return km
```

La función devuelve el resultado de evaluar la expresión a continuación de **return**.

```
print(50, 'millas equivalen a', millas_a_kilometros(), 'kilómetros')
```

Funciones

def *NOMBRE*:
BLOQUE

NOMBRE es el nombre de la función.
BLOQUE es un bloque de código.

BLOQUE se ejecuta cada vez que la función *NOMBRE* es invocada.

Ejemplo:

```
def millas_a_kilometros():  
    mi = 50  
    km = mi * 1.60934  
    return km
```

```
print(50, 'millas equivalen a', millas_a_kilometros(), 'kilómetros')
```



Invocación a la función.

Funciones

def *NOMBRE*:
BLOQUE

NOMBRE es el nombre de la función.
BLOQUE es un bloque de código.

BLOQUE se ejecuta cada vez que
la función *NOMBRE* es invocada.

Ejemplo:

```
def millas_a_kilometros():  
    mi = 50  
    km = mi * 1.60934  
    return km
```

```
print(50, 'millas equivalen a', millas_a_kilometros(), 'kilómetros')
```

En Python las funciones tienen que declararse antes de ser invocadas (antes == más arriba).

Funciones

Programa 1

```
def millas_a_kilometros():  
    mi = 50  
    km = mi + 1.60934  
    return km
```

Los errores quedan encapsulados en un solo lugar. Es más fácil localizarlos y corregirlos.

```
print(50, 'millas equivalen a', millas_a_kilometros(), 'kilómetros')  
...  
print(50, 'millas equivalen a', millas_a_kilometros(), 'kilómetros')
```

Programa 2

```
mi = 50  
km = mi + 1.60934  
print(mi, 'millas equivalen a', km, 'kilómetros')  
...  
mi = 50  
km = mi + 1.60934  
print(mi, 'millas equivalen a', km, 'kilómetros')
```

Funciones

Ejercicio:

Eliminar la redundancia en el siguiente programa:

```
print('Ana: ¡Yo tengo escalera!')
i = 1
while i < 6:
    print(i, end=' ')
    i = i + 1
print()
print('Agustín: ¡Yo también!')
i = 1
while i < 6:
    print(i, end=' ')
    i = i + 1
print()
print('Pablo: ¡Mirá qué casualidad!')
i = 1
while i < 6:
    print(i, end=' ')
    i = i + 1
print()
```

```
Ana: ¡Yo tengo escalera!
1 2 3 4 5
Agustín: ¡Yo también!
1 2 3 4 5
Pablo: ¡Mirá qué casualidad!
1 2 3 4 5
```

Parámetros

Un elefante se balanceaba
sobre la tela de una araña
Como veía que resistía
fueron a llamar a otro elefante.

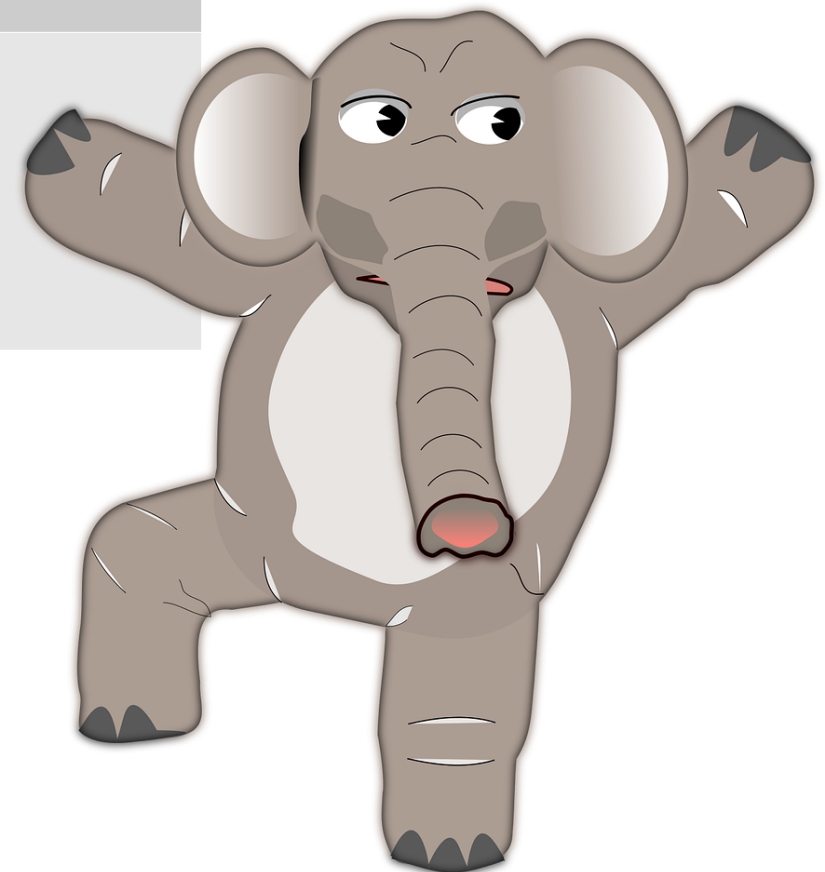
Cuatro elefantes se balanceaban
sobre la tela de una araña
Como veían que resistía
fueron a llamar a otro elefante.

Dos elefantes se balanceaban
sobre la tela de una araña
Como veían que resistía
fueron a llamar a otro elefante.

Cinco elefantes se balanceaban
sobre la tela de una araña
Como veían que resistía
fueron a llamar a otro elefante.

Tres elefantes se balanceaban
sobre la tela de una araña
Como veían que resistía
fueron a llamar a otro elefante.

Seis elefantes se balanceaban
sobre la tela de una araña
Como veían que resistía
fueron a llamar a otro elefante.



Parámetros

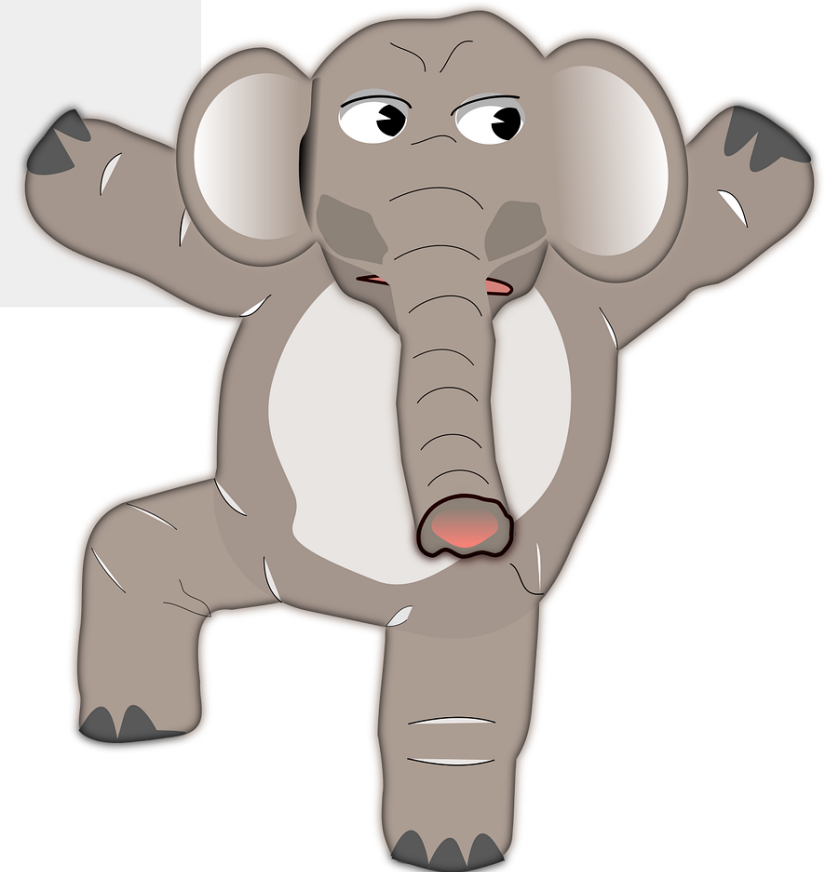
Canción:

Un elefante se balanceaba
sobre la tela de una araña
Como veía que resistía
fueron a llamar a otro elefante.

[Estribillo [Dos](#)]
[Estribillo [Tres](#)]
[Estribillo [Cuatro](#)]
[Estribillo [Cinco](#)]
[Estribillo [Seis](#)]

Estribillo ([cantidad](#)):

[cantidad](#) elefantes se balanceaban
sobre la tela de una araña
Como veían que resistía
fueron a llamar a otro elefante.



Parámetros

Ejemplo:

```
def millas_a_kilometros(mi):  
    km = mi * 1.60934  
    return km
```

Se pueden incluir cualquier cantidad de **parámetros** separados por comas. ¡Usar nombres descriptivos!

```
print(50, 'millas equivalen a', millas_a_kilometros(50), 'kilómetros')  
print(63, 'millas equivalen a', millas_a_kilometros(63), 'kilómetros')  
print(84, 'millas equivalen a', millas_a_kilometros(84), 'kilómetros')  
print(17, 'millas equivalen a', millas_a_kilometros(17), 'kilómetros')  
print(99, 'millas equivalen a', millas_a_kilometros(99), 'kilómetros')
```

...

Parámetros

Ejemplo:

```
def millas_a_kilometros(mi):  
    km = mi * 1.60934  
    return km
```

Dentro de la función se usan del mismo modo que las variables.

```
print(50, 'millas equivalen a', millas_a_kilometros(50), 'kilómetros')  
print(63, 'millas equivalen a', millas_a_kilometros(63), 'kilómetros')  
print(84, 'millas equivalen a', millas_a_kilometros(84), 'kilómetros')  
print(17, 'millas equivalen a', millas_a_kilometros(17), 'kilómetros')  
print(99, 'millas equivalen a', millas_a_kilometros(99), 'kilómetros')
```

...

Parámetros

Ejemplo:

```
def millas_a_kilometros(mi):  
    km = mi * 1.60934  
    return km
```

Se invoca a la función usando diferentes **argumentos**.

```
print(50, 'millas equivalen a', millas_a_kilometros(50), 'kilómetros')  
print(63, 'millas equivalen a', millas_a_kilometros(63), 'kilómetros')  
print(84, 'millas equivalen a', millas_a_kilometros(84), 'kilómetros')  
print(17, 'millas equivalen a', millas_a_kilometros(17), 'kilómetros')  
print(99, 'millas equivalen a', millas_a_kilometros(99), 'kilómetros')
```

...

Parámetros

Ejemplo:

```
def millas_a_kilometros(mi):  
    km = mi * 1.60934  
    return km
```

Usando parámetros se construyen soluciones generales que permiten resolver un gran número de problemas particulares.

```
print(50, 'millas equivalen a', millas_a_kilometros(50), 'kilómetros')  
print(63, 'millas equivalen a', millas_a_kilometros(63), 'kilómetros')  
print(84, 'millas equivalen a', millas_a_kilometros(84), 'kilómetros')  
print(17, 'millas equivalen a', millas_a_kilometros(17), 'kilómetros')  
print(99, 'millas equivalen a', millas_a_kilometros(99), 'kilómetros')
```

...

Parámetros

Ejercicio:

Escribir una función `primeros_numeros(n)` en Python que, dado un número n , imprima por pantalla los números del 1 al n en una única línea.

Ejemplos de invocaciones y el resultado esperado:

`primeros_numeros(1)`

1

`primeros_numeros(3)`

1 2 3

`primeros_numeros(6)`

1 2 3 4 5 6

`primeros_numeros(10)`

1 2 3 4 5 6 7 8 9 10

Observaciones finales

```
def raiz_cuadrada(n):  
    i = 1  
    while i * i <= n:  
        i = i + 1  
    return i - 1  
  
x = 1  
while x <= 5:  
    print(raiz_cuadrada(x))  
    x = x + 1
```

Cada ejecución de una función tiene su **propio espacio de memoria**, como si fuera otro programa.

n, **i** son **alcanzables** dentro de **raiz_cuadrada**, pero no fuera.

Observaciones finales

```
def raiz_cuadrada(n) :  
    i = 1  
    while i * i <= n:  
        i = i + 1  
    return i - 1
```

```
def potencia(a,n) :  
    rv = 1  
    i = 0  
    while i < n:  
        rv = rv * a  
        i = i + 1  
    return rv
```

$i \neq i$ y $n \neq n$

Observaciones finales

Ejercicio: (opcional)

a) Definir en Python una función `es_bisiesto(anio)` que devuelva `True` si el año `anio` es bisiesto y `False` en caso contrario.

Nota: un año es bisiesto tanto si es divisible por 400 como si es divisible por 4 pero no por 100.

b) Definir en Python una función `imprimir_bisisestos(desde, hasta)` que imprima en la consola todos los años entre `desde` y `hasta` que sean bisiestos.

Repaso de la clase de hoy:

- Estructuras de control de flujo: condicional y ciclo
- Modularidad del código: definición de funciones

Próximos temas:

- Listas
- Manejo de archivos

Para la próxima clase:

- Resolver las secciones 2 y 3 de la guía de ejercicios