

Curso de nivelación de algoritmos

Algoritmo

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

Ejemplo

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

- 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,
- 9.- fin

Instrucción

Una **instrucción** es una operación que:

- **transforma los datos**, 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**,
- 9.- **fin**

Programa y Variables

Un **programa** es una implementación de un algoritmo en un lenguaje de programación.

Una **variable** es un nombre que denota una **dirección de memoria** en la que se almacena un valor, que puede ser leído o modificado.

Estado

Se denomina **estado** al valor de todas las variables de un programa en un punto de su ejecución, más la siguiente instrucción a ejecutar.

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

Asignación

VARIABLE = EXPRESIÓN

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

Ejemplo:

a = 0

b = [1, 2, 3]

b[a] = 1

b[b[0]] = 999 * b[a] + a

b[b[1]/333-1] = 123;

Secuencialización

PROG1; PROG2

PROG1 y *PROG2* son programas. Se ejecuta primero *PROG1*; una vez finalizado, se ejecuta *PROG2*.

Estado

Se denomina **estado** al valor de todas las variables de un programa en un punto de su ejecución, más la siguiente instrucción a ejecutar.

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

Ejemplo de repaso

`a = 0`

`b = []`

`b.append(1)`

`b.append(999 * b[a] + a)`

`b[b[1]/333-2] = 123`

Ejemplo de repaso

$\{ a=\uparrow \wedge b=\uparrow \}$

`a = 0`

`b = []`

`b.append(1)`

`b.append(999 * b[a] + a)`

`b[b[1]/333-2] = 123`

Ejemplo de repaso

$\{ a=\uparrow \wedge b=\uparrow \}$

`a = 0`

$\{ a=0 \wedge b=\uparrow \}$

`b = []`

`b.append(1)`

`b.append(999 * b[a] + a)`

`b[b[1]/333-2] = 123`

Ejemplo de repaso

$\{ a=\uparrow \wedge b=\uparrow \}$

`a = 0`

$\{ a=0 \wedge b=\uparrow \}$

`b = []`

$\{ a=0 \wedge \text{len}(b)=0 \}$

`b.append(1)`

`b.append(999 * b[a] + a)`

`b[b[1]/333-2] = 123`

Ejemplo de repaso

$\{ a=\uparrow \wedge b=\uparrow \}$

a = 0

$\{ a=0 \wedge b=\uparrow \}$

b = []

$\{ a=0 \wedge \text{len}(b)=0 \}$

b.append(1)

$\{ a=0 \wedge \text{len}(b)=1 \wedge b[0]=1 \}$

b.append(999 * b[a] + a)

b[b[1]/333-2] = 123

Ejemplo de repaso

$\{ a=\uparrow \wedge b=\uparrow \}$

a = 0

$\{ a=0 \wedge b=\uparrow \}$

b = []

$\{ a=0 \wedge \text{len}(b)=0 \}$

b.append(1)

$\{ a=0 \wedge \text{len}(b)=1 \wedge b[0]=1 \}$

b.append(999 * b[a] + a)

$\{ a=0 \wedge \text{len}(b)=2 \wedge b[0]=1 \wedge b[1]=999 \}$

b[b[1]/333-2] = 123

Ejemplo de repaso

$\{ a=\uparrow \wedge b=\uparrow \}$

a = 0

$\{ a=0 \wedge b=\uparrow \}$

b = []

$\{ a=0 \wedge \text{len}(b)=0 \}$

b.append(1)

$\{ a=0 \wedge \text{len}(b)=1 \wedge b[0]=1 \}$

b.append(999 * b[a] + a)

$\{ a=0 \wedge \text{len}(b)=2 \wedge b[0]=1 \wedge b[1]=999 \}$

b[b[1]/333-2] = 123

$\{ a=0 \wedge \text{len}(b)=2 \wedge b[0]=1 \wedge b[1]=123 \}$

Condicional

if *CONDICION*: *PROG1*

CONDICION es una expresión que arroja resultado verdadero o falso; *PROG1* es un programa.

PROG1 se ejecuta **si y sólo si** *CONDICION* arroja valor verdadero.

Ejemplo:

```
if 1 > 5:  
    print '1 es mayor que 5.'  
if 1 < 5:  
    print '1 es menor que 5.'
```


Condicional

if *CONDICION: PROG1*

Ejemplo:

```
a = 10
```

```
b = [100, 1]
```

```
if b[0] / (a * 10) == b[1]:
```

```
    b[0] = b[0] - 1
```

```
    b[1] = b[1] * 5
```

```
print a, b[0], b[1]
```

Condicional

if *CONDICION: PROG1*

Ejemplo:

```
a = 10
```

```
b = [100, 1]
```

```
if b[0] / (a * 10) == b[1]:
```

```
    b[0] = b[0] - 1    # bloque
```

```
    b[1] = b[1] * 5    # indentado
```

```
print a, b[0], b[1]
```

```
# imprime: 10 99 5
```

Condicional

if *CONDICION*: *PROG1* else: *PROG2*

CONDICION es una expresión que arroja V o F.

PROG1 y *PROG2* son programas.

PROG1 se ejecuta **sii** *CONDICION* arroja valor verdadero.

PROG2 se ejecuta **sii** *CONDICION* arroja valor falso.

Condicional

if *CONDICION: PROG1* else: *PROG2*

Ejemplo:

```
a = 'a'
b = 'b'
if a > b:
    a = '$'
else:
    a = '%'
print a
```

Ciclo

while *CONDICION*: *PROG1*

CONDICION es una expresión que arroja resultado verdadero o falso; *PROG1* es un programa.

PROG1 se repite **mientras** *CONDICION* arroje valor V.

Ejemplo:

```
i = 0
while i < 3:
    print i,
    i = i + 1
print
```

Ciclo

while *CONDICION: PROG1*

Ejemplo:

```
i = 0
while i < 3:
    if i % 2 == 0:
        print i, 'es par'
    else:
        print i, 'es impar'
    i = i + 1
```

Ciclos anidados

Ejemplo:

```
fil = 1
while fil <= 5:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil + 1
```

Programa

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

- Asignación
 - ***VARIABLE = EXPRESIÓN***
- Condicional
 - ***if CONDICION: PROG1 else: PROG2***
- Ciclo
 - ***while CONDICION: PROG1***

Acerca de Python

Python es un lenguaje de alto nivel muy versátil.
Nos permite escribir código de **muchas** formas distintas.

Recomendamos (por ahora) usar sólo los elementos de programación que les enseñamos.

Acerca de Python

Python es un lenguaje de alto nivel muy versátil.
Nos permite escribir código de **muchas** formas distintas.

Recomendamos (por ahora) usar sólo los elementos de programación que les enseñamos.

Ejemplo: Armar la lista [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

```
cuadrados = []  
x = 0  
while x < 10:  
    cuadrados.append(x*x)  
    x = x + 1
```

```
cuadrados = []  
for x in range(10):  
    cuadrados.append(x*x)
```

```
def cuad(x): return x*x  
cuadrados = map(cuad, range(10))
```

```
cuadrados = [x*x for x in range(10)]
```

...

Programa

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

- Asignación
 - ***VARIABLE = EXPRESIÓN***
- Condicional
 - ***if CONDICION: PROG1 else: PROG2***
- Ciclo
 - ***while CONDICION: PROG1***

Volviendo a nuestro ejemplo...

```
fil = 1
while fil <= 5:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil + 1
```

Output: ?

Volviendo a nuestro ejemplo...

```
fil = 1
while fil <= 5:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil + 1
```

Output:

```
|
| 2
| 2 3
| 2 3 4
| 2 3 4 5
```

Volviendo a nuestro ejemplo...

```
fil = 1
while fil <= 5:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil + 1
```

?

Output:

```
|
| 2
| 2 3
| 2 3 4
| 2 3 4 5
```

Output:

```
| 2 3 4 5
| 2 3 4
| 2 3
| 2
|
```

Volviendo a nuestro ejemplo...

```
fil = 1
while fil <= 5:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil + 1
```

Output:

```
|
| 2
| 2 3
| 2 3 4
| 2 3 4 5
```

```
fil = 5
while fil >= 1:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil - 1
```

Output:

```
| 2 3 4 5
| 2 3 4
| 2 3
| 2
|
```

Volviendo a nuestro ejemplo...

```
fil = 1
while fil <= 5:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil + 1
```

?

Output:

```
|
| 2
| 2 3
| 2 3 4
| 2 3 4 5
```

Output:

```
|
|
| 2 3
| 2 3
| 2 3 4 5
| 2 3 4 5
```


Volviendo a nuestro ejemplo...

```
fil = 1
while fil <= 5:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil + 1
```

Output:

```
|
| 2
| 2 3
| 2 3 4
| 2 3 4 5
```

```
fil = 1
while fil <= 5:
    if fil % 2 != 0:
        col = 1
        while col <= fil:
            print col,
            col = col + 1
        print
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil + 1
```

Output:

```
|
|
|23
|23
|2345
|2345
```

¿Cómo podemos hacer para reusar este código sin tener que copiarlo una y otra vez?

```
fil = 1
while fil <= 5:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil + 1
```

```
fil = 5
while fil >= 1:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil - 1
```

```
fil = 1
while fil <= 5:
    if fil % 2 != 0:
        col = 1
        while col <= fil:
            print col,
            col = col + 1
        print
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil + 1
```

Output:

```
|
| 2
| 2 3
| 2 3 4
| 2 3 4 5
```

Output:

```
| 2 3 4 5
| 2 3 4
| 2 3
| 2
|
```

Output:

```
|
|
|23
|23
|2345
|2345
```

Funciones

```
def imprimir_fila(fil):  
    col = 1  
    while col <= fil:  
        print col,  
        col = col + 1  
    print
```

Ejemplos:

`imprimir_fila(2)` imprime: **1 2**

`imprimir_fila(5)` imprime: **1 2 3 4 5**

Funciones

```
fil = 1
while fil <= 5:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil + 1
```

Output:

```
|
| 2
| 2 3
| 2 3 4
| 2 3 4 5
```

```
fil = 5
while fil >= 1:
    col = 1
    while col <= fil:
        print col,
        col = col + 1
    print
    fil = fil - 1
```

Output:

```
| 2 3 4 5
| 2 3 4
| 2 3
| 2
|
```

```
fil = 1
while fil <= 5:
    if fil % 2 != 0:
        col = 1
        while col <= fil:
            print col,
            col = col + 1
        print
        col = 1
        while col <= fil:
            print col,
            col = col + 1
        print
    fil = fil + 1
```

Output:

```
|
|
| 2 3
| 2 3
| 2 3 4 5
| 2 3 4 5
```

Funciones

```
def imprimir_fila(fil):  
    col = 1  
    while col <= fil:  
        print col,  
        col = col + 1  
    print
```

```
fil = 1  
while fil <= 5:  
    imprimir_fila(fil)  
    fil = fil + 1
```

Output:

```
|  
| 2  
| 2 3  
| 2 3 4  
| 2 3 4 5
```

```
fil = 5  
while fil >= 1:  
    imprimir_fila(fil)  
    fil = fil - 1
```

Output:

```
| 2 3 4 5  
| 2 3 4  
| 2 3  
| 2  
|
```

```
fil = 1  
while fil <= 5:  
    if fil % 2 != 0:  
        imprimir_fila(fil)  
        imprimir_fila(fil)  
    fil = fil + 1
```

Output:

```
|  
|  
| 2 3  
| 2 3  
| 2 3 4 5  
| 2 3 4 5
```

Funciones

```
def imprimir_fila(fil):  
    col = 1  
    while col <= fil:  
        print col,  
        col = col + 1  
    print
```

```
fil = 1  
while fil <= 5:  
    imprimir_fila(fil)  
    fil = fil + 1
```

```
fil = 5  
while fil >= 1:  
    imprimir_fila(fil)  
    fil = fil - 1
```

```
fil = 1  
while fil <= 5:  
    if fil % 2 != 0:  
        imprimir_fila(fil)  
        imprimir_fila(fil)  
    fil = fil + 1
```

Resultado: Código modular.

- Más claro para los humanos.
- Más fácil de actualizar.

¿Qué pasa si quiero separar los números con “,” en lugar de “ ”?

Función

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.

Procedimiento

```
def imprimir_filas(fil):  
    col = 1  
    while col <= fil:  
        print col,  
        col = col + 1  
    print
```

Procedimiento == Función que no devuelve valor alguno.

Función

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

Ejemplos:

```
raiz_cuadrada(8) devuelve: 2  
raiz_cuadrada(9) devuelve: 3  
raiz_cuadrada(10) devuelve: 3
```

```
#!/usr/bin/python

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.

Un detalle técnico

En Python, las funciones deben definirse **antes** de ser usadas.

- Antes == Más arriba en el archivo de código.
- Por eso, el código principal suele definirse abajo de todo.

```
#!/usr/bin/python

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
```

Repaso de la clase de hoy

- Condicional: if.. ; if..else..
- Ciclo: while..; ciclos anidados.
- Modularidad del código: funciones.
- Alcance (scope) de variables.

Temas de la clase que viene

- Complejidad algorítmica.
- Algoritmos de búsqueda.
- Algoritmos de ordenamiento.