



Técnica: ITERACIÓN



Repetición de procesos

- Uso muy común de las computadoras: repetición de procesos para crear soluciones (llamados también ciclos o bucles)
- Técnicas de programación para repetir procesos
 - Iteración
 - Recursión



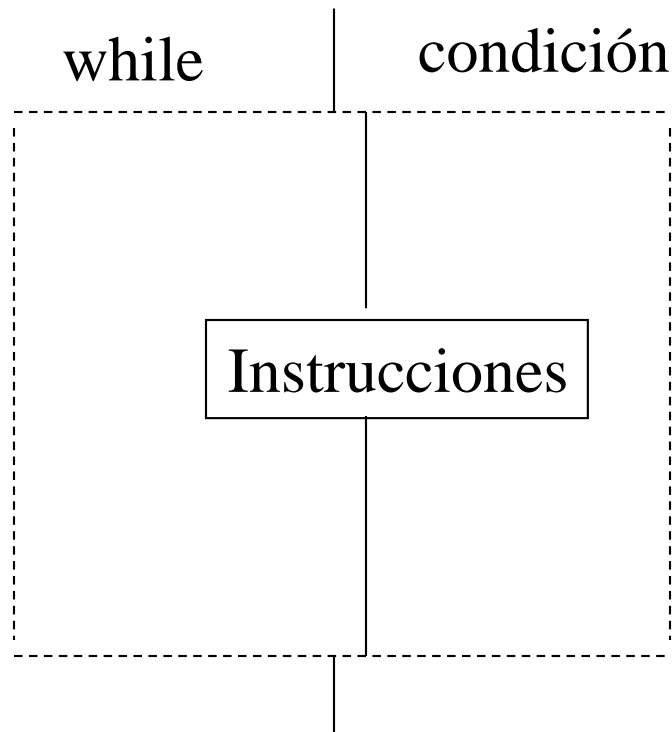
Técnica de iteración

- Iteración: se origina del verbo iterar que significa repetir
- Los lenguajes de programación brindan varias estructuras para implementar esta técnica
- Python ofrece dos estructuras de iteración
 - **while**
 - **for**

Estructura *while* (mientras)

- Repite la ejecución de un bloque de instrucciones mientras una condición sea verdadera

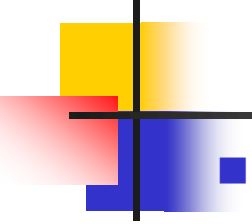
Representación del
while mediante un
diagrama de flujo





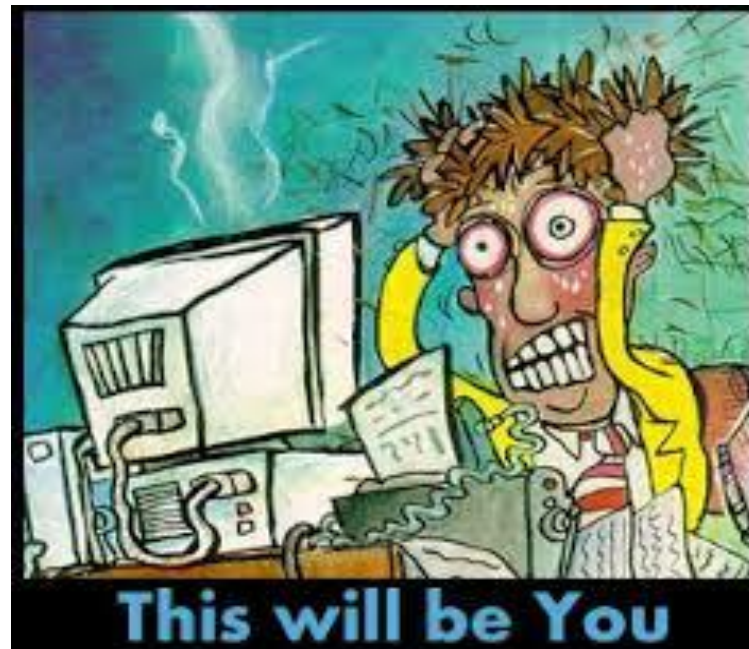
¿ cómo funciona ?

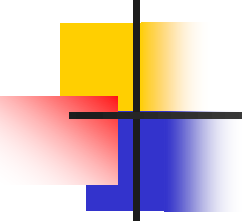
- Evaluar la condición: verdadera o falsa
- Si la condición es verdadera (True)
 - Ejecuta el bloque de instrucciones dentro del while
 - Al terminar de ejecutar ese bloque regresa al inicio del ciclo (línea donde esta el while) para evaluar nuevamente la condición
- Si la condición es falsa (False)
 - No ejecuta el bloque de instrucciones dentro del while
 - La ejecución continúa en la siguiente instrucción que se encuentre después de la estructura del while

- 
- Responsabilidad del programador: asegurar la propiedad de finito, el ciclo debe terminar
 - Normalmente en el bloque de instrucciones del while se modifican variables de la condición para que ésta llegue a ser falsa en algún momento y termine la ejecución del bloque repetitivo



- Si la condición nunca llega a ser falsa tendremos un “programa ciclado”



- 
-
- En el bloque de instrucciones puede haber cualquier tipo de instrucción, incluyendo condicionales, iteración, asignación, etc.
 - Ciclo dentro de otro ciclo: ciclo anidado

- 
-
- Sintaxis de la estructura en Python

while condición:
Instrucciones

- El bloque de instrucciones del while está indentado respecto a la palabra while

Funciones iterativas con números

Obtener la cantidad de dígitos

- Ilustrar técnica de iteración con datos numéricos
- Función dígitos: ejemplos del comportamiento de la función

```
>>> digitos(742)
```

```
3
```

```
>>> digitos(45278)
```

```
5
```



Metodología para desarrollar programas

- Paso 1: entender el problema

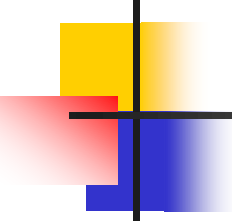
- Definir

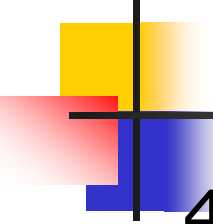
- Entradas: número entero positivo
 - Salidas: cantidad de dígitos de ese número
 - Proceso: contar la cantidad de dígitos. Vimos algunos ejemplos.
 - Restricciones: por ahora no vamos a validar las restricciones



Paso 2: diseñar un algoritmo

- Hay dos operadores aritméticos que ayudan en el desarrollo de algoritmos de descomposición de números
 - Operador de residuo: %
 - El residuo 10 sirve para obtener el dígito menos significativo de un número
>>> 45278 % 10
8
 - Operador de división entera: // (usado en este algoritmo)
 - La división entera entre 10 sirve para eliminar el dígito menos significativo de un número
>>> 45278 // 10
4527

- 
- Este algoritmo en particular usa `// 10` para ir eliminando el dígito menos significativo de un número
 - Proceso repetitivo:
 - Analizar cada dígito del número de entrada: elimina el dígito menos significativo
 - Uso de una variable tipo **contador**: por cada dígito eliminado contamos 1 en esa variable
 - Fin del proceso repetitivo:
 - Cuando se eliminen todos los dígitos del número analizado
 - Al final del proceso repetitivo la variable tipo contador tendrá el resultado



$$\begin{array}{r}
 45278 \mid 10 \\
 8 \mid 4527 \mid 10 \\
 7 \mid 452 \mid 10 \\
 2 \mid 45 \mid 10 \\
 5 \mid 4 \mid 10 \\
 4 \mid 0
 \end{array}$$

Por cada división que hicimos contamos 1 más a los dígitos eliminados: 5 divisiones.

(En este caso los residuos no fueron importantes pero considere ese operador para la solución de otros problemas)

IMPORTANTE: ¿ Cuándo se termina el ciclo ? Cuando se eliminen todos los dígitos del número: el cociente queda en cero



Paso 3: codificar el algoritmo

- # Función: dígitos (versión 1.0: solución inicial)
- # Entradas: número entero
- # Salidas: cantidad de dígitos que tiene el número de entrada
- # Restricciones: en esta versión del programa no se validan

```
def digitos(n):  
    d = 0          # variable tipo contador: para contar cantidad de dígitos  
    while n != 0:  
        n = n // 10  
        d = d + 1  
    return d
```



■ Paso 4: probar y evaluar el programa

```
>>> digitos(45278)
5
```

Corrida manual:

| Variables: | <u>n</u> | <u>d</u> |
|------------|----------|----------|
| | 45278 | 0 |
| | 4527 | 1 |
| | 452 | 2 |
| | 45 | 3 |
| | 4 | 4 |
| | 0 | 5 |

Seguimiento o corrida manual:

*** Lista de variables de forma horizontal**

*** Cambios en variables de forma vertical**

- 
- Este programa hay que modificarlo para considerar restricciones y casos especiales

- Hay casos en los que se obtienen resultados incorrectos:

NO SE VALIDAN LAS RESTRICCIONES:

LA ENTRADA DEBE SER UN NÚMERO ENTERO

```
>>> digitos(789.123)
```

```
3
```

```
>>> digitos("abc")
```

Traceback (most recent call last):

```
File "<pyshell#35>", line 1, in <module>  
    digitos("abc")
```

```
File "<pyshell#34>", line 4, in digitos  
    n = n // 10
```

TypeError: unsupported operand type(s) for //: 'str' and 'int'



CUANDO HAY CASOS ESPECIALES:

```
>>> digitos(0)
```

```
0
```

VERIFICACIÓN DE RESTRICCIONES: tipos de datos y valores de datos

VALIDAR TIPOS DE DATOS

- Función predefinida: **isinstance**
- Función booleana (retorna True o False) que determina si el tipo de datos de un objeto es igual al tipo de datos indicado
- Sintaxis
isinstance (objeto, **tipo de datos**) → True
False



■ Ejemplos

```
>>> a, b, c, d = 10, "hola", True, 10.0
```

```
>>> isinstance(a, int)
```

```
True
```

```
>>> isinstance(d, int)
```

```
False
```

```
>>> isinstance(d, float)
```

```
True
```

```
>>> isinstance(a, float)
```

```
False
```

```
>>> isinstance(b, str)
```

```
True
```

```
>>> isinstance(c, bool)
```

```
True
```

```
>>> isinstance(b, int)
```

```
False
```

***VALIDAR:
TIPOS
DE
DATOS***



VALIDAR VALORES DE DATOS: if

- Validar que un dato esté en un rango
 - Ejemplo: número entre 1 y 999
- Validar que un dato tenga valores específicos
 - Ejemplo: códigos de carrera
 - "IC": Ingeniería en Computación
 - "ATI": Administración de TI

***VALIDAR:
VALORES
DE
DATOS***



CASOS ESPECIALES

- Condiciones en las cuales no aplican las generalidades aplicadas a los datos del programa produciendo resultados incorrectos
- Ocupan un tratamiento o lógica diferente
- En un algoritmo pueden existir varios casos especiales: el algoritmo los debe identificar y solucionar



Función dígitos

Caso especial:

- El caso del valor 0 como entrada es un caso especial, ya que es igual a la condición de terminación del ciclo, por eso inicialmente el resultado nos da 0. Este caso especial lo manejamos específicamente para que de el valor correcto que es 1

Función: dígitos (versión 2.0: valida restricciones y casos especiales)
Entradas: número entero
Salidas: cantidad de dígitos que tiene el número de entrada
Restricciones: número entero positivo

```
def digitos(n):  
    if isinstance(n, int) == False:    # validar que entrada sea un entero  
        return "ERROR: DATO DE ENTRADA DEBE SER UN ENTERO"  
    if n < 0:                          # validar que numero >= 0  
        return "ERROR: DATO DE ENTRADA DEBE SER UN POSITIVO"  
    if n == 0:                        # caso especial: dato de entrada es cero  
        return 1  
    d = 0                             # variable tipo contador: para contar cantidad de dígitos  
    while n != 0:  
        n = n // 10  
        d = d + 1  
    return d
```




Función para multiplicar dos números naturales usando sumas sucesivas

■ Ejemplos del comportamiento de la función

>>> multiplica(3, 4)

12 $\rightarrow 3 + 3 + 3 + 3$

>>> multiplica(8, 3)

24 $\rightarrow 8 + 8 + 8$

>>> multiplica(5, 0)

0 \rightarrow no hay sumas



Metodología para desarrollar programas

- Paso 1: entender el problema

- Definir

- Entradas: dos números naturales (0, 1, 2, ...), multiplicando y multiplicador
 - Salidas: producto o multiplicación de esos números
 - Proceso: obtener el producto de ambos números sumando el multiplicando la cantidad de veces indicadas por el multiplicador. Vimos algunos ejemplos.
 - Restricciones: números naturales



■ Paso 2: diseñar un algoritmo

- Las sumas sucesivas es un proceso repetitivo:
 - Uso de una variable tipo **acumulador**: en esta variable vamos acumulando el valor del multiplicando tantas veces como indique el multiplicador
 - Uso de una variable tipo **contador**: por cada suma contamos 1
- Fin del proceso repetitivo:
 - Cuando se hayan realizado todas las sumas: el contador de sumas llegó a la cantidad de veces indicada por el multiplicador
- La variable tipo acumulador tendrá el resultado

■ Paso 3: codificar el algoritmo

Funcion: multiplica dos números usando el método de sumas sucesivas
Entradas: el multiplicando y el multiplicador
Salidas: producto o multiplicación de los números de entrada
Restricciones: números naturales (0, 1, 2, ...)

```
def multiplica(multiplicando, multiplicador):  
    # validar que entradas sean numeros naturales  
    if isinstance(multiplicando, int) == False or \  
        isinstance(multiplicador, int) == False:  
        return "ERROR: LAS ENTRADAS DEBEN SER NUMEROS NATURALES"  
    if multiplicando < 0 or multiplicador < 0:  
        return "ERROR: LAS ENTRADAS DEBEN SER NUMEROS NATURALES"  
    # proceso de multiplicacion  
    producto = 0 # variable tipo acumulador: sumas del multiplicando  
    cont = 0 # variable tipo contador: las veces que se ha sumado el multiplicando  
    while cont < multiplicador:  
        producto = producto + multiplicando  
        cont = cont + 1  
    return producto
```



Paso 4: probar y evaluar el programa

```
>>> multiplica(5, 3)
```

```
15
```

Corrida manual:

| Variables: | <u>multiplicando</u> | <u>multiplicador</u> | <u>producto</u> | <u>cont</u> |
|------------|----------------------|----------------------|-----------------|-------------|
| | 5 | 3 | 0 | 0 |
| | | | 5 | 1 |
| | | | 10 | 2 |
| | | | 15 | 3 |



Función booleana (retorna True o False) para determinar si un entero tiene un dígito par

- Paso 1: entender el problema

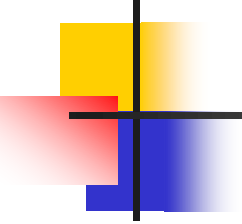
- Definir

- Entradas: un número entero
 - Salidas: es una función booleana, retorna un valor booleano, True o False
 - Proceso: algoritmo numérico que retorna True si el valor de entrada contiene un dígito par y False de lo contrario
 - Restricciones: validamos que la entrada sea un entero



■ Paso 2: desarrollar algoritmo

- Proceso repetitivo:
 - Analizar cada dígito del número
 - Obtener el menos significativo por medio del operador aritmético %
 - Uso de una variable tipo **indicador (flag, bandera, centinela)**: variables a las que el programador asigna valores específicos bajo su control para dirigir el flujo de ejecución
 - En este caso la variable tipo indicador se usa para controlar la ejecución del proceso repetitivo y a la vez da el resultado final
 - Uso de una variable tipo **contador**: por cada dígito eliminado contamos 1 en esa variable

- 
-
- Condiciones para el fin del proceso repetitivo:
 - Cuando se encuentre un dígito par
 - Cuando se eliminen todos los dígitos del número analizado

Paso 3: codificar el algoritmo

Funcion: determinar si en un número natural hay un dígito par

Entradas: número natural

Salidas: True si el valor de entrada tiene un dígito par, False de lo contrario

Restricciones: número natural

```
def tiene_par(num):
    if not isinstance(num, int) or num < 0:    # Restricciones
        return "Error: dato debe ser un número natural"
    if num == 0:    # Caso especial
        return True
    par = False    # Variable tipo indicador para determinar el resultado, inicialmente no hay pares
    while num != 0 and par == False:
        if num % 2 == 0:
            par = True
        else:
            num = num // 10
    return par
```

Paso 4: probar y evaluar el programa

```
>>> tiene_par(125)
```

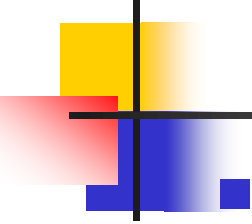
```
True
```

```
>>> tiene_par("abc")
```

```
'Error: dato debe ser un número entero'
```

```
>>> tiene_par(11)
```

```
False
```



Función para obtener el término n-ésimo de la sucesión de Fibonacci

■ Paso 1: entender el problema

- En matemáticas la sucesión de Fibonacci es la sucesión infinita de números naturales:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Término 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...

Término 1: 0

Término 2: 1

Término n-ésimo: suma los dos términos anteriores
(término $n-1$ + término $n-2$)

Hay que desarrollar un programa que calcule el término n-ésimo de esta sucesión



■ Definir

- Entradas: número de término que se va a calcular
- Salidas: término respectivo
- Proceso: calcular el término n-ésimo de la sucesión de Fibonacci según se describió.

Ejemplos del funcionamiento:

$$\text{fib}(1) \rightarrow 0$$

$$\text{fib}(5) \rightarrow 3$$

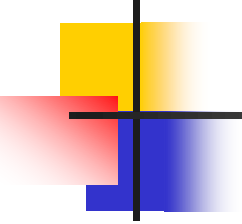
$$\text{fib}(9) \rightarrow 21$$

- Restricciones: entero ≥ 1



Paso 2: desarrollar un algoritmo

- Por definición están dados los dos primeros términos: 0, 1
- Proceso repetitivo consiste en calcular el siguiente término basado en los dos anteriores
 - Usar una variable tipo **contador** para ir llevando la cantidad de términos que se han calculado.
 - Usar dos variables tipo **temporales (intermedias, auxiliares)**: variables que contienen valores que ayudan a obtener el resultado final
 - Variable con el término trasanterior ($n - 2$)
 - Variable con el término anterior ($n - 1$)
 - Usar una variable tipo **acumulador** para el resultado
 - Variable con el término actual (suma del anterior y el trasanterior)

- 
-
- Condición para terminar el while
 - Cuando hayamos encontrado el término requerido: sucede cuando el contador de términos calculados llegue el término que se ocupa
 - Restricciones
 - El valor de entrada es un número entero (≥ 1)

Paso 3: codificar el algoritmo

Funcion: calcular el n-esimo término de la sucesión de Fibonacci

Entradas: entero

Salidas: término de la sucesión de Fibonacci

Restricciones: numero de entrada debe ser un entero mayor a 0

```
def fib(n):
    if not isinstance(n, int): # restricciones
        return "Error: dato debe ser un entero"
    if n < 1:
        return "Error: término debe ser 1 o mas"
    if n == 1:                # caso especial: termino 1
        return 0
    if n == 2:                # caso especial: termino 2
        return 1
    trasanterior = 0 # calcular términos
    anterior = 1
    contador = 2 # cantidad de términos
    while contador < n:
        contador = contador + 1
        actual = anterior + trasanterior
        trasanterior = anterior
        anterior = actual
    return actual
```

Paso 4: probar y evaluar el programa

```
>>> fib(1)
```

```
0
```

```
>>> fib(5)
```

```
3
```



Estatuto *break*

Finaliza inmediatamente la ejecución del ciclo (estatuto de iteración) donde se encuentre

- El flujo de ejecución sale del ciclo y continúa en la primera instrucción después del ciclo
- Ejemplo:

while condición:

...

if condición:

 break

...



Estatuto *continue*

- Regresa al inicio del ciclo (estatuto de iteración) donde se encuentre para continuar con la siguiente iteración
- Ejemplo:

while condición:

...

if condición:

continue

...



Práctica

- Siguiendo la metodología de desarrollo de programas construya la función **tabla_multiplicar**. Recibe tres números enteros (≥ 0) e imprime la tabla de multiplicar del primer argumento empezando en el segundo argumento y terminando en el tercer argumento. Validar restricciones, incluyendo que el segundo argumento debe ser \leq al tercer argumento. Ejemplo del funcionamiento:

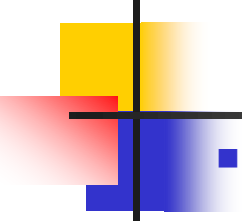
```
>>> tabla_multiplicar(12, 3, 6)
```

```
12 x 3 = 36
```

```
12 x 4 = 48
```

```
12 x 5 = 60
```

```
12 x 6 = 72
```

- 
- Siguiendo la metodología de desarrollo de programas construya la función **factorial**. El factorial de un número n ($n!$) está definido solo para enteros positivos. Este cálculo es el producto de todos los números enteros desde 1 hasta n :

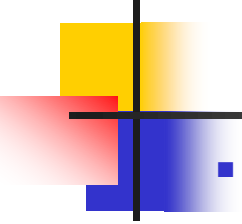
$$n! = 1 * 2 * 3 * \dots * (n - 1) * n$$

Por definición: $0! = 1$

Ejemplos del funcionamiento:

| | | |
|--------------|-------|-----------------------------------|
| factorial(0) | → 1 | |
| factorial(1) | → 1 | |
| factorial(4) | → 24 | (cálculo: $1*2*3*4$ o $4*3*2*1$) |
| factorial(6) | → 720 | (calculo: $1*2*3*4*5*6$) |

La función recibe un número entero (≥ 0) y retorna su factorial.
Validar restricciones.

- 
- Siguiendo la metodología de desarrollo de programas construya la función booleana **primo**. Retorna True si el número es primo y False de lo contrario. Un número natural ≥ 2 es primo si tiene dos divisores: 1 y él mismo. Para practicar algoritmos con iteración implemente el proceso de divisiones sucesivas desde 1 hasta el número inclusive. Por cada división con residuo 0 suma 1 a un contador de divisores. Al final de las divisiones el contador debe ser igual a 2 para que sea un primo. La función recibe un número entero (≥ 2), valide esas restricciones.

Ejemplos del funcionamiento:

```
>>> primo(1)
```

```
False
```

```
>>> primo(17)
```

```
True
```

```
>>> primo(1000)
```

```
False
```

Posteriormente puede implementar otros algoritmos eficientes para determinar si un número es primo.