Programación imperativa

Repaso

Paradigma de programación

Un paradigma de programación se refiere a un enfoque o estilo particular de desarrollo de software que establece un conjunto de principios y reglas para diseñar y construir programas de computadora. Representa una *manera de pensar y abordar la resolución de problemas* utilizando conceptos, estructuras y técnicas específicas.

- Paradigma funcional
  En este paradigma, el énfasis está en las f*unciones como elementos fundamentales*. Se enfoca en el cálculo de resultados basados en la evaluación de funciones y evita los cambios de estado y las mutaciones de datos (transparencia referencial).
- Paradigma imperativo
  Este paradigma se centra en describir cómo se deben ejecutar las instrucciones y cómo se modifican los datos en el proceso. Los programas se construyen a través de una *secuencia de instrucciones* y se enfocan en *cambiar el estado de las variables*.

Diferencias entre el paradigma funcional y el imperativo

1. **Cambio de estado:** En el paradigma imperativo, se enfoca en el cambio de estado de las variables y cómo se modifican a lo largo del programa. Se utilizan instrucciones y sentencias para actualizar y manipular los datos. En contraste, en el paradigma funcional, se evita el cambio de estado y se enfoca en la evaluación de funciones. Los datos se consideran inmutables, y en lugar de modificarlos, se crean nuevos valores a partir de los existentes.
2. **Funciones y procedimientos:** En el paradigma imperativo, los programas se construyen mediante la secuencia de instrucciones que indican cómo realizar una serie de pasos para lograr un resultado. Se utilizan procedimientos o subrutinas para agrupar y reutilizar bloques de código. En el paradigma funcional, el énfasis está en las funciones como elementos fundamentales. Las funciones se tratan como ciudadanos de primera clase y pueden ser pasadas como argumentos, retornadas como resultados y almacenadas en variables, de manera que el programa se construye con composiciones de funciones.
3. **Recursión**: La recursión es ampliamente utilizada en el paradigma funcional para resolver problemas mediante la autollamada de funciones. Se enfatiza la recursión estructural, donde los problemas se dividen en subproblemas más pequeños hasta alcanzar un caso base. En el paradigma imperativo, la recursión también se puede utilizar, pero en muchos casos se prefiere la iteración. La iteración utiliza bucles para repetir un bloque de código, mientras que la recursión utiliza llamadas recursivas para repetir el bloque de código resolviendo subproblemas más pequeños. La iteración se controla mediante una variable de control y tiene una estructura lineal,

mientras que la recursión se controla mediante una condición de finalización y tiene una estructura más ramificada.

4. **Estado y efectos secundarios**: Los efectos secundarios se refieren a cualquier modificación o interacción con el entorno fuera de la función en sí misma. Esto incluye cambios en variables globales, impresiones en pantalla, lectura/escritura de archivos u otras interacciones con el sistema. En el paradigma funcional, se busca que las funciones sean puras, lo que significa que su resultado depende únicamente de sus entradas y no tienen efectos imprevistos en otras partes del sistema. Las funciones puras no modifican variables externas ni producen cambios en el estado global del programa. Esto permite tener un mayor control sobre el flujo y el comportamiento del programa, facilita el razonamiento y la comprensión del código, y reduce la posibilidad de errores sutiles. El paradigma imperativo se centra en el cambio de estado y cómo se modifican los datos a lo largo del programa. Los efectos secundarios son una parte integral de este enfoque y se utilizan para lograr el comportamiento deseado del programa.
5. **Variables y su manipulación**: En el paradigma imperativo, las variables se utilizan para representar información mutable y su valor puede cambiar durante la ejecución del programa. En el paradigma funcional, las variables se utilizan para representar valores inmutables y su valor se mantiene constante a lo largo del programa. En lugar de modificar directamente una variable, se crean nuevas variables con valores actualizados en función de las necesidades de procesamiento.
6. **Pensamiento declarativo y pensamiento imperativo**: La principal diferencia entre el pensamiento declarativo y el pensamiento imperativo radica en el nivel de abstracción y el enfoque en la especificidad de los pasos. El pensamiento declarativo se enfoca en la descripción del problema y la especificación de los resultados deseados, sin entrar en detalles sobre cómo se debe lograr. Se centra en la lógica y las relaciones entre los elementos del problema. Por otro lado, el pensamiento imperativo se centra en los pasos y procedimientos específicos que se deben seguir para resolver el problema, detallando cómo se deben realizar las acciones. El pensamiento declarativo se enfoca en el "qué" y se abstrae de los detalles de implementación, mientras que el pensamiento imperativo se enfoca en el "cómo" y describe los pasos detallados para lograr un resultado.

Paradigma de programación funcional	**Paradigma imperativo**
Enfoque en la evaluación de funciones	Enfoque en el cambio de estado
Datos inmutables	Datos mutables
Énfasis en la recursión	Uso de iteración y bucles
Evita o minimiza los efectos secundarios	Acepta y utiliza efectos secundarios
Funciones como elementos fundamentales	Instrucciones y sentencias como elementos fundamentales
Programas construidos mediante la composición de funciones	Programas construidos mediante la secuencia de instrucciones
Menor énfasis en la manipulación directa de variables	Manipulación directa de variables

Mayor énfasis en la reutilización de funciones	Mayor énfasis en la reutilización de procedimientos
Enfoque en el pensamiento declarativo	Enfoque en el pensamiento imperativo

Programación imperativa

Novedades:

- Nueva operación: la asignación (cambia el valor de una variable).
- Nuevo mecanismo de repetición: iteración.
- Nuevo tipo de datos: el arreglo.
  - Secuencia de valores de un tipo (como las listas).
  - Longitud prefijada.
  - Acceso directo a una posición (en las listas, hay que acceder primero a las anteriores)
- Las funciones NO pertenece a un tipo de datos

Lenguaje Python

Python soporta varios paradigmas de programación, por lo tanto se usará un subconjunto de todo lo que abarca Python con tal de tratar la programación imperativa.

Características del lenguaje:

- **Es un lenguaje interpretado:** el código fuente se ejecuta directamente sin la necesidad de compilarlo previamente. Es decir, en lugar de ser traducido a un código de máquina específico, el código fuente se interpreta línea por línea en tiempo de ejecución por un programa llamado intérprete: el intérprete lee cada instrucción del código fuente y la ejecuta de inmediato. No se genera un archivo ejecutable separado como en el caso de los lenguajes compilados.
- **Tiene tipado dinámico:** los tipos de datos son verificados en tiempo de ejecución, es decir, durante la ejecución del programa. En un lenguaje de tipado dinámico, las variables no están asociadas a un tipo de datos específico en el momento de su declaración, y su tipo puede cambiar *durante la ejecución del programa*. Sin embargo, en esta materia lo vamos a pensar con tipado estático: Declararemos el tipo de cada variable en tiempo de diseño.
  - Tipado dinámico: Una variable puede tomar valores de distintos tipos.

    ```
    def suma (x,y):
        res = x + y
        return res
    ```

  - Tipado estático: La comprobación de tipificación se realiza durante la compilación (y no durante la ejecución)

    - ```
      def suma (x: int, y:int) -> int:
          res: int = x + y
          return res
      ```

- **Es fuertemente tipado:** *Una vez que una variable toma un tipo, ya no puede cambiar*. Python es fuertemente tipado porque impone restricciones en las operaciones entre tipos incompatibles para garantizar la integridad y seguridad del programa.

Programa Python:

- **Colección de tipos y funciones.**

```
def nombreFunción (parámetros) -> tipoResultado
    bloqueInstrucciones
```

- **Su evaluación consiste en ejecutar una por una las instrucciones del bloque.**
- **El orden entre las instrucciones es de arriba a abajo.**

Características de la Programación Imperativa

Variables

- Es un nombre asociado a un espacio en memoria.
- Puede tomar distintos valores a lo largo de la ejecución.
- **En Python** se declaran dando su nombre (y opcionalmente su tipo):
    - `x: int`
    - `c: char`

Instrucciones

- Asignación.
    - Es la operación fundamental para modificar el valor de una variable.
        - `variable = expresión`
    - Es una operación asimétrica.
        - Lado izquierdo: debe ir una variable u otra expresión que represente una posición en memoria.
        - Lado derecho: una expresión del mismo tipo que la variable.
    - Al evaluar la expresión de la derecha se obtiene un valor, el cual se copia en el espacio de memoria de la variable, sin afectar el resto de la memoria.
- Condicional (if ... else ...).
- Ciclos (while ...).
- Procedimientos: funciones que no devuelven valores pero modifican sus argumentos.
```

- Retorno de control (con un valor, return).
    - Termina la ejecución de una función.
    - Retorna el control a su invocador.
    - Devuelve el valor de la expresión como resultado.

Transformación de estados

Llamamos **estado** de un programa a los valores de todas sus variables en un punto de su ejecución: antes de ejecutar la primera instrucción, entre dos instrucciones, después de ejecutar la última instrucción.

La ejecución de un programa se trata de una **sucesión de estados. **Mediante la asignación se transforman estados y el resto de instrucciones son de control, es decir, modifican el flujo (orden) de ejecución.

```
def ejemplo () -> int :
    x:int = 0 //Estado 1
    x = x + 3 //Estado 2
    x = 2 * x //Estado 3
    return x
```

A cada estado se le puede dar un nombre, que representará el conjunto de valores de las variables entre dos instrucciones de un programa.

```
instruccion
//estado nombre_estado
otra instruccion
```

Luego de nombrar un estado, podemos referirnos al valor de una variable en dicho estado.

`nombre_variable@nombre_estado`

Ejecución simbólica

La ejecución simbólica es una técnica utilizada en el campo de la verificación y análisis de programas que consiste en realizar un seguimiento simbólico de las operaciones y valores en lugar de utilizar valores concretos. En lugar de ejecutar el programa con datos reales, se utilizan símbolos que representan variables y se realizan cálculos simbólicos en lugar de aritmética real.

```
def suc(x: int) -> int:
        //estado a;
    x = x + 2
        //estado b
        //vale x == x@a+2;
        «En el estado b, x vale lo que valía en el estado a más 2»
    x = x - 1
        //estado c
        //vale x == x@b-1;
        «En el estado c, x vale lo que valía en el estado b menos 1»
    return x
```

De esta manera, mediante la transformación de estados, podremos realizar una ejecución simbólica del programa, declarando cuánto vale cada variable, en cada estado del programa, en función de los valores anteriores.

Los argumentos de entrada de las funciones

Para indicar que una función recibe argumentos de entrada usamos variables. Estas variables toman valor cuando el llamador invoca a la función.

En los lenguajes imperativos, en general, existen dos tipos de pasajes de parámetros:

- Pasaje por valor: Se crea una copia local de la variable dentro de la función.
- Pasaje por referencia: Se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera.

Pasajes de argumentos en lenguajes de programación

Valor y Referencia

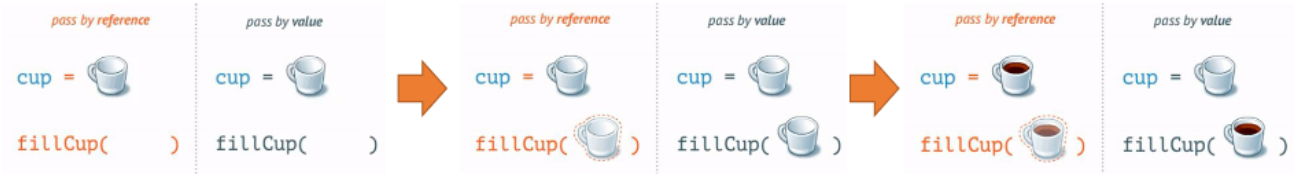
En el contexto de la programación, un "**valor**" se refiere a un dato o información concreta almacenada en memoria. Puede ser un número, un carácter, una cadena de texto u otro tipo de dato primitivo. Los valores se pueden asignar a variables, manipular y utilizar en operaciones dentro de un programa. Por otro lado, una "**referencia**" es un tipo de dato que apunta a una ubicación específica en memoria donde se encuentra el valor real. En lugar de almacenar el valor directamente, una referencia almacena la dirección de memoria donde se encuentra el valor. La referencia actúa como un "**puntero**" que permite acceder y modificar el valor real almacenado en esa ubicación de memoria.

En cuanto a los pasajes, se refiere a **cómo se transmiten** los datos entre diferentes partes de un programa, como funciones, métodos o subrutinas. Hay dos formas comunes de pasar datos: por valor y por referencia.

- **Paso por valor**: En el paso por valor, se **copia el valor** de la variable original en una **nueva ubicación de memoria** y **se pasa esa copia** a la función o método. Cualquier modificación realizada dentro de la función no afectará a la variable original, ya que se está trabajando con una

copia independiente. Los cambios realizados en la copia solo existen dentro del ámbito de la función. Esto se utiliza para tipos de datos primitivos y objetos inmutables.

- **Paso por referencia:** En el paso por referencia, en lugar de copiar el valor, **se pasa una referencia (dirección de memoria)** a la función o método. Esto significa que las modificaciones realizadas dentro de la función **afectarán directamente a la variable original**, ya que ambas están apuntando al mismo dato en memoria. Los cambios realizados dentro de la función son visibles fuera del ámbito de la función. Esto se utiliza principalmente para estructuras de datos complejas o grandes, como arreglos o objetos, donde copiar todo el contenido podría ser ineficiente.



Programación Imperativa

- Funciones y procedimientos: ambos ejecutan un grupo de sentencias. Las funciones devuelven un valor y los procedimientos no.
- Existen 3 tipos de pasajes de parámetros:
  - Entrada (in): Al salir de la función o procedimiento, la variable pasada como parámetro continuará teniendo su valor original.
  - Salida (out): Al salir de la función o procedimiento, la variable pasada como parámetro tendrá un nuevo valor asignado dentro de dicha función o procedimiento. Su valor inicial no importa ni debería ser leído dentro de la función o procedimiento en cuestión.
  - Entrada y salida (inout): Al salir de la función o procedimiento, la variable pasada como parámetro tendrá un nuevo valor asignado dentro de dicha función o procedimiento. Su valor inicial SÍ importa dentro de la función o procedimiento en cuestión.

Pasaje de argumentos en Python

- Conceptualmente, el comportamiento va a depender del tipo de datos de la variable. Los tipos primitivos (int, char, string, etc) se pasan por valor. Los tipos compuestos y estructuras (Ej: listas) se pasan por referencia.
- Técnicamente todos los parámetros son por referencia siempre y las variables de tipos primitivos tienen referencias a valores inmutables.

Extensión de la Especificación

Pasaje de parámetros: in, out e inout - Funciones y Procedimientos

problema *nombre*(*parámetros*) : tipo de dato del resultado (*opcional*){
requiere *etiqueta*: { condiciones sobre los parámetros de entrada }
modifica: parametros que se modificarán
asegura *etiqueta*: { condiciones sobre los parámetros de salida }
Si x es un parametro inout, x@pre se refiere al valor que tenía x al entrar a la función }

- *nombre*: nombre que le damos al problema
  - será resuelto por una función con ese mismo nombre
- *parámetros*: lista de parámetros separada por comas, donde cada parámetro contiene:
  - Tipo de pasaje (entrada: **in**, salida: **out**, entrada y salida: **inout**)
  - Nombre del parámetro
  - Tipo de datos del parámetro o una **variable de tipo**
- *tipo de dato del resultado*: tipo de dato del resultado del problema (inicialmente especificaremos funciones) o una **variable de tipo**
  - En los asegura, podremos referenciar el valor devuelto con el nombre de **res**
- El tipo de dato del resultado se vuelve opcional, pues ahora podremos especificar programas que no devuelvan resultados, sino que sólo modifiquen sus parámetros.

En python:

problema *duplicar*(*inout* $x : seq\langle T \rangle$){
  modifica: x
  asegura: {x tendrá todos los elementos de x@pre y además, se le concatenará otra copia de x@pre a continuación.}
  asegura: {x tendrá el doble de longitud que x@pre.}
}

```
def duplicar(x: list):
  x *= 2
```

```
def duplicar(valor: str, referencia: list):
    valor *= 2
    print("Dentro de la función duplicar: str: " + valor)
    referencia *= 2
    print("Dentro de la función duplicar:  referencia: " + str(referencia))


x: str = "abc"
y: list = ['a', 'b', 'c']
print("ANTES: ")
print("x: " + x)
print("y: " + str(y))
duplicar(x, y)
print("DESPUES: ")
print("x: "  + x)
print("y: " + str(y))
```

```
ANTES:
x: abc
y: ['a', 'b', 'c']
Dentro de la función duplicar: str: abcabc
Dentro de la función duplicar:  referencia: ['a', 'b', 'c', 'a', 'b', 'c']
DESPUES:
x: abc
y: ['a', 'b', 'c', 'a', 'b', 'c']
```