title: "Práctica 2"

date: "2025-05-31"

summary: "El objetivo de esta práctica es identificar elementos esenciales de los lenguajes de programación en Python, a diferencia de la práctica anterior que se centró en C. Se analizará cómo Python maneja conceptos como nombres, marcos de activación, bloques de alcance y administración de memoria. Un aspecto clave a destacar es el tipado dinámico de Python y su gestión automática de la memoria, lo cual simplifica la programación en comparación con el control manual que ofrece C."

Universidad Autónoma de Baja California

Facultad de Ingeniería, Arquitectura y Diseño

Paradigmas de la programación

Práctica 2

Elementos básicos de los lenguajes de programación

Arturo Rafael Cornejo Escobar

31 de abril del 2025

INTRODUCCIÓN

El objetivo de esta práctica es identificar los **elementos esenciales de los lenguajes de programación**, incluyendo: **nombres**, **marcos de activación**, **bloques de alcance**, **administración de memoria**, **expresiones**, **comandos**, **control de secuencia** (selección, iteración y recursión), **subprogramas** y **tipos de datos** en el ámbito de Python a diferencia de la práctica pasada que era de C.

Por lo que es importante marcar las diferencias en los elementos:

Los **nombres** manejan un tipado dinámico, lo que se entiendo como qué el tipo de una variable se es determinado durante el tiempo de la ejecución del programa y no durante la compilación, lo que abre la posibilidad de cambiar el tipo según el dato agregado.

En el caso de los **marcos de activación**, son manejados de forma automática también, esto infiere que cuando se llama a una función o sub programa, Python crea el entorno en la pila para las variables locales y las referencias, además de limpiarlo al finalizar, mientras que en C es un manejo más manual sobre la gestión de la pila.

La primera diferencia que se puede notar, son los **bloques de alcance** ya que, están delimitados por la identación a diferencia de definirlo por llaves ("{ }") y los **comandos** se terminan con una nueva línea y agrupados según la identación.

DESARROLLO

Nombres

Los nombres nos sirven para identificar diferentes elementos con etiquetas que nosotros asignamos, estos elementos son: **Clases, funciones, variables, y constantes.**

- **Nombres de clases**: `Genre`, `Book`, `DigitalBook`, `Member`, `Library`
- **Nombres de métodos**: `add_book`, `display_books`, `issue_book`, `to_dict`, `from_dict`
- **Nombres de variables**: `book_id`, `title`, `author`, `publication_year`, `genre`, `quantity`, `file_format`, `member_id`, `name`, `issued_books`, `choice`, `filename`
- **Constantes**: `FICTION`, `NON_FICTION`, `SCIENCE`, `HISTORY`, `FANTASY`, `BIOGRAPHY`, `OTHER` (dentro de la clase `Genre`)

Marcos de Activación

Cuando se llama a una función o método, se crea un **marco de activación** (o *stack frame*) en la pila de llamadas del programa. Este marco almacena información sobre la ejecución de la función, incluyendo:

- **Variables locales**: Por ejemplo, en `main()`, `library`, `book_id`, `title`, `author`, etc., son locales a esa llamada de función.
- **Parámetros**: En `add_book(self, book)`, `self` y `book` son parámetros pasados al método.
- **Dirección de retorno**: La ubicación en el código a la que se debe regresar después de que la función termine.

Cada vez que se llama a un método como `library.add_book(book)` o `book.to_dict()`, un nuevo marco de activación se ingresa en la pila.

Bloques de Alcance

El **alcance** se refiere a la región del código donde un nombre es reconocido y puede ser accedido. Python utiliza **alcance léxico**, lo que significa que el alcance se determina por la ubicación física del código.

- **Alcance de clase**: Las variables y métodos definidos dentro de una clase (por ejemplo, `self.id` en `Book`) son accesibles dentro de las instancias de esa clase.

```
class Book:
    def __init__(self, book_id, title, author, publication_year, genre, quantity):
        self.id = book_id             # 'self.id' es un atributo de instancia
        self.title = title            # 'self.title' es un atributo de instancia
        self.author = author          # 'self.author' es un atributo de instancia
        self.publication_year = publication_year
        self.genre = genre
        self.quantity = quantity
```

- **Alcance de método**: Las variables definidas dentro de un método (por ejemplo, `data` en `DigitalBook.to_dict()`) son locales a ese método y no se pueden acceder fuera de él.

```
class DigitalBook(Book):
    def to_dict(self):
        data = super().to_dict() # 'data' es una variable local del método
to_dict()
        data["file_format"] = self.file_format
        return data
```

- **Alcance global**: Aunque no se utiliza explícitamente para muchas variables en este ejemplo específico, cualquier variable definida en el nivel superior del módulo tendría alcance global.
- **Alcance de función**: Las variables definidas dentro de una función (por ejemplo, `choice` en `main()`) son locales a esa función.

```
def main():
    library = Library() # 'library' es una variable local de la función main()
    while True:
        choice = int(input("Indica tu opcion: ")) # 'choice' es una variable local
de la función main()
```

Administración de memoria

Python utiliza la **administración automática de memoria** a través de un proceso llamado **recolección de basura** (*garbage collection*). Los objetos se asignan en el **montón** (*heap*) cuando se crean (por ejemplo, `Book()`, `Member()`, `Library()`). El módulo `memory_management`tiene la intención de rastrear las asignaciones y desasignaciones del **heap**.

El método `__init__` en `Book`, `Member` y `Library` llama explícitamente a `memory_management.increment_heap_allocations(1)`, indicando que se está creando un objeto y se está asignando memoria. De manera similar, el método `__del__`, que es un destructor, llama a `memory_management.increment_heap_deallocations(1)` cuando un objeto está a punto de ser recolectado por el recolector de basura, lo que implica que la memoria para ese objeto está siendo liberada.

Clase `MemoryManagement`

La clase `MemoryManagement` tiene los siguientes componentes:

- **`__init__(self)`**: Este es el constructor de la clase. Inicializa dos atributos:
  - `self.heap_allocations`: Un contador que rastrea la cantidad total de "memoria" asignada en el heap. Se inicializa en 0.
  - `self.heap_deallocations`: Un contador que rastrea la cantidad total de "memoria" desasignada del heap. Se inicializa en 0.

```
def  __init__(self):

    self.heap_allocations  =  0
```

```
    self.heap_deallocations  =  0
```

- `increment_heap_allocations(self, size)`: Este método se utiliza para aumentar el contador de asignaciones. Cuando se crea un nuevo objeto en el programa principal que consume memoria del heap, se llamaría a este método, pasando el `size` (tamaño) de la memoria asignada.

```
def  increment_heap_allocations(self, size):

    '''Increment heap allocations'''

    self.heap_allocations  +=  size
```

- `increment_heap_deallocations(self, size)`: Este método se utiliza para aumentar el contador de desasignaciones. Cuando un objeto es destruido o su memoria es liberada, se llamaría a este método, pasando el `size` de la memoria desasignada.

```
def  increment_heap_deallocations(self, size):

    '''Increment heap deallocations '''

    self.heap_deallocations  +=  size
```

- `display_memory_usage(self)`: Este método simplemente imprime el estado actual de las asignaciones y desasignaciones del heap. Muestra los valores de `self.heap_allocations` y `self.heap_deallocations`.

```
def  display_memory_usage(self):

    '''Display memory usage'''

    print(f"Heap allocations: {self.heap_allocations} bytes")

    print(f"Heap deallocations: {self.heap_deallocations} bytes")
```

Al final del módulo, se crea una instancia de la clase `MemoryManagement` llamada `memory_management`:

Esta instancia global permite que otros módulos) accedan y actualicen los contadores de asignación y desasignación de memoria sin tener que crear una nueva instancia de `MemoryManagement` cada vez.

Expresiones

Las **expresiones** son combinaciones de valores, variables, operadores y llamadas a funciones que se evalúan a un único valor.

- **Expresiones aritméticas**: `book.quantity -= 1`
- **Expresiones booleanas**: `book and member and book.quantity > 0`, `is_digital == 's'`
- **Expresiones de cadena**: `f"ID libro: {book.id}"` (*f-string*)
- **Construir lista a partir de expresión en secuencia**: `[book.to_dict() for book in self.books]`

Comandos

Los **comandos** (o sentencias) son instrucciones que realizan una acción. No necesariamente devuelven un valor.

- **Asignación**: `self.id = book_id`
- **Llamadas a métodos**: `library.add_book(book)`
- **Sentencias de impresión**: `print("\nEl libro fue agregado exitosamente!\n")`
- **Sentencias de control de flujo**: `if`, `elif`, `else`, `while`, `for`

Control de Flujo

Selección

Las sentencias `if`/`elif`/`else` controlan el flujo de ejecución basándose en condiciones.

- En `main()`, el bloque `if choice == 1:` determina qué acción realizar según la entrada del usuario.
- En `issue_book()`, `if book and member and book.quantity > 0:` verifica si se puede prestar un libro.
- En `load_library_from_file()`, `try-except` maneja posibles `FileNotFoundError`.

Iteración

Los bucles se utilizan para ejecutar repetidamente un bloque de código.

- **Bucle `while True:` en `main()`**: Muestra continuamente el menú hasta que el usuario elige salir.
- **Bucles `for`**: Se utilizan para iterar sobre listas de libros o miembros (por ejemplo, `for book in self.books:` en `display_books()`).

Subprogramas

Los **subprogramas** se refieren a funciones y métodos, que son bloques de código diseñados para realizar una tarea específica y pueden ser reutilizados.

- **Métodos**: Funciones definidas dentro de clases, que operan sobre los datos del objeto (por ejemplo, `add_book`, `to_dict`).
- **Métodos estáticos**: Métodos que pertenecen a la clase pero no operan sobre datos de instancia (por ejemplo, `Book.from_dict`). Se definen con `@staticmethod`.
- **Métodos de clase**: Métodos que operan sobre la clase misma, (por ejemplo, `Genre.all_genres`). Se definen con `@classmethod`.
- **Funciones**: Bloques de código independientes (por ejemplo, `main()`).

Tipos de Datos

Python es un lenguaje de tipado dinámico, lo que significa que no se declaran explícitamente los tipos de datos.

- `int` **(Enteros)**: Para números sin decimales como IDs (`book_id`, `member_id`), años (`publication_year`), cantidades (`quantity`) y opciones de menú (`choice`).
- `str` **(Cadenas de Texto)**: Para texto como títulos (`title`), autores (`author`), géneros (`genre`), nombres (`name`), formatos de archivo (`file_format`) y nombres de archivos (`filename`).
- `list` **(Listas)**: Para colecciones ordenadas de elementos, como la lista de todos los libros (`self.books`), todos los miembros (`self.members`) o los IDs de los libros prestados por un miembro (`self.issued_books`).
- `dict` **(Diccionarios)**: Para colecciones de pares clave-valor. Se usan para representar objetos (`Book`, `Member`, etc.) como estructuras de datos que se pueden guardar en JSON y cargar desde él.
- `bool` **(Booleanos)**: Para valores de verdad (`True` o `False`), como la variable `is_digital` que indica si un libro es digital o el resultado de condiciones lógicas (`if book and member`).

Referencias

Cooper, K. D., & Torczon, L. (2012). **The procedure abstraction**. En *Elsevier eBooks* (pp. 269-330). https://doi.org/10.1016/b978-0-12-088478-0.00006-2

Alcance. (s. f.). JSvis. https://centrogeo.github.io/JSvis/13-Alcance.html

TylerMSFT. (s. f.). **Nombres representativos**. Microsoft Learn. https://learn.microsoft.com/es-es/cpp/build/reference/decorated-names?view=msvc-170

IBM Debug for z/OS. (s. f.). **Expresiones en C**. IBM. https://www.ibm.com/docs/es/debug-for-zos/15.0.x?topic=programs-c-c-expressions

---

Dirección del repositorio:

https://github.com/roixarturo/portafolio1

Dirección de la página de GitHub:

https://roixarturo.github.io/portafolio1/