

# Funciones y Clases

## Funciones `j_eval` y `j_plot` en R

```
# #| include: false
source("./docs/j_eval_j_plot.r")
```

### Introducción

Hasta ahora, hemos escrito código que se ejecuta de arriba hacia abajo, como una lista de mercado. Pero en proyectos reales de geomática, necesitamos reutilizar lógica. Si tienes un cálculo complejo para medir distancias o para valuar un predio, no quieres copiar y pegar ese código cien veces.

Para eso existen las **funciones** (máquinas que procesan datos) y las **clases** (moldes para crear objetos complejos). En este capítulo aprenderás a encapsular tu lógica para que tu código sea modular, limpio y fácil de mantener.

### Objetivos de aprendizaje

- Crear funciones reutilizables para automatizar cálculos espaciales recurrentes.
- Manejar parámetros opcionales, valores por defecto y argumentos dinámicos.
- Entender los principios básicos de la Programación Orientada a Objetos (POO) para modelar elementos del mundo real.

### 1. Funciones: bloques de código reutilizables

Una función es como una receta de cocina: tú le pasas los ingredientes (parámetros de entrada), ella realiza un proceso interno (cuerpo de la función) y te entrega un resultado (valor de retorno).

Para ilustrar el poder de las funciones, vamos a construir tres herramientas fundamentales para cualquier analista espacial. Cada una nos enseñará un concepto nuevo de programación:

1. **La calculadora matemática (`haversine`):** Resolveremos el cálculo de la distancia entre dos ciudades. Como la Tierra no es plana, no podemos usar una simple línea recta (Pitágoras).
2. **El procesador por lotes (`medir_ruta`):** Crearemos una función que reciba una lista completa de coordenadas, la recorra con un ciclo `for` y llame a nuestra primera función (`haversine`) repetidas veces para medir los tramos de una ruta.

3. **El recolector dinámico (describir\_punto):** Crearemos una función que use **argumentos dinámicos** (\*\*kwargs en Python, ... en R y kwargs... en Julia) para atrapar cualquier cantidad de variables extra (clima, población, etc.) que el usuario decida enviarnos.

### La matemática detrás de nuestra calculadora espacial

La **fórmula de Haversine** calcula la distancia de círculo máximo (*great-circle distance*) (Figure 1) entre dos puntos en la superficie de una esfera, como la Tierra, a partir de sus coordenadas de latitud y longitud. La fórmula tiene en cuenta la curvatura de la esfera, lo que la hace mucho más precisa que una simple distancia euclíadiana (línea recta) para aplicaciones geoespaciales.

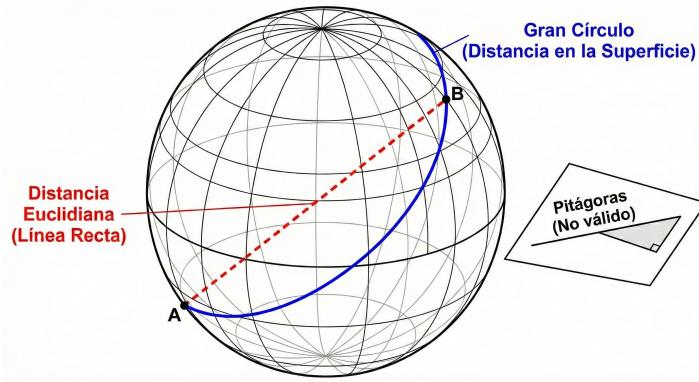


Figure 1: Distancia de Círculo Máximo - fórmula de Haversine

Matemáticamente, adaptada a las variables de nuestro código, la ecuación general se ve así:

$$distancia = 2 \cdot radio \cdot \arcsin \left( \sqrt{\sin^2 \left( \frac{lat2 - lat1}{2} \right) + \cos(lat1) \cdot \cos(lat2) \cdot \sin^2 \left( \frac{lon2 - lon1}{2} \right)} \right)$$

Donde:

- *distancia*: Distancia física entre los dos puntos a lo largo de la curva de la esfera.
- *radio*: Radio de la esfera de referencia (para la Tierra, aproximadamente 6371.0 km).
- *lat1*, *lat2*: Latitudes del punto de origen y destino, estrictamente en radianes.
- *lon1*, *lon2*: Longitudes del punto de origen y destino, estrictamente en radianes.

*El truco del radio:* El cálculo trigonométrico interno solo produce un ángulo sin unidad física (radianes). Para convertirlo en distancia real, lo multiplicamos por el radio de la Tierra. Si

metemos el radio en kilómetros (6371.0), sale en kilómetros; si lo metemos en millas (3958.8), sale en millas.

En programación, solemos dividir esta gran fórmula en partes más pequeñas (las variables `dlat`, `dlon`, `a` y `c`) para facilitar la lectura del código y evitar errores, reemplazando el arccsin por la función `atan2` que es computacionalmente más estable.

En la práctica, para que el computador procese esta gran ecuación sin ahogarse (y para evitar errores de paréntesis), los programadores la dividen en tres pasos secuenciales usando las variables `a`, `c` y `distancia`:

1. **El ajuste esférico (`a`):** Calcula el cuadrado de la mitad de la cuerda recta entre los dos puntos.

$$a = \sin^2\left(\frac{dlat}{2}\right) + \cos(lat1) \cdot \cos(lat2) \cdot \sin^2\left(\frac{dlon}{2}\right)$$

2. **El ángulo central (`c`):** Usa la función arcotangente (`atan2`) para hallar el ángulo exacto en radianes desde el centro de la Tierra.

$$c = 2 \cdot \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right)$$

3. **La distancia física (`distancia`):** Convierte el ángulo en una longitud real multiplicándolo por el radio.

$$distancia = radio \cdot c$$

**i** Nota técnica: ¿Por qué usamos `atan2` en lugar de `arcsin` en el código?

Si buscas la fórmula matemática clásica de Haversine en un libro, notarás que utiliza la función arcoseno (`arcsin`). Sin embargo, en nuestro código de programación usamos la función arcotangente (`atan2`). ¿Por qué esta diferencia?

Todo se reduce a la **precisión computacional (punto flotante)**. Cuando dos coordenadas están muy cerca la una de la otra, usar  $\text{arcsin}(\sqrt{a})$  puede generar imprecisiones severas de redondeo en el procesador.

Para solucionarlo, la programación aprovecha la trigonometría básica ( $\tan = \frac{\sin}{\cos}$ ) usando la función `atan2(y, x)`:

- Sabiendo que  $a = \sin^2$ , entonces el seno (cateto opuesto,  $y$ ) es  $\sqrt{a}$ .
- Por la regla pitagórica ( $\sin^2 + \cos^2 = 1$ ), el coseno (cateto adyacente,  $x$ ) es  $\sqrt{1-a}$ .

Al ingresar  $c = 2 \cdot \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right)$ , obligamos al computador a calcular el ángulo exacto usando ambos lados del triángulo, garantizando mediciones perfectas ya sea que midamos la distancia entre dos continentes o entre dos pasos en la calle.

Veamos cómo se programan estas máquinas paso a paso, explicando cada línea de su funcionamiento:

## Python

```
# #| eval: false

# Importamos la librería matemática nativa de Python para usar senos, cosenos y radianes
import math

# --- FUNCIÓN 1: CÁLCULO MATEMÁTICO BÁSICO ---
def haversine(lat1, lon1, lat2, lon2, radio=6371.0):
    # RECIBE: 4 números (lat/lon de dos puntos) y un radio opcional (por defecto 6371.0)

    # Convierte la diferencia de latitudes a radianes (el idioma de los computadores)
    dlat = math.radians(lat2 - lat1)

    # Convierte la diferencia de longitudes a radianes
    dlon = math.radians(lon2 - lon1)

    # Calcula 'a': el cuadrado de la mitad de la cuerda recta entre los puntos
    a = (math.sin(dlat / 2) ** 2 +
         math.cos(math.radians(lat1)) * math.cos(math.radians(lat2)) * math.sin(dlon / 2) **

    # Calcula 'c': la distancia angular central usando la función arcotangente (atan2)
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

    # SACÁ: La multiplicación del ángulo por el radio, dando la distancia física real
    return radio * c

print("--- Cálculo de distancia simple ---")
```

```
--- Cálculo de distancia simple ---
```

```
# Ejecutamos la función. Como no le damos el radio, usa el defecto en km.
distancia_km = haversine(4.6097, -74.0817, 6.2442, -75.5812)
print(f"Distancia Bogotá-Medellín: {distancia_km:.2f} km")
```

Distancia Bogotá-Medellín: 246.14 km

```
# Sobrescribimos el parámetro 'radio' para obtener el resultado en millas (3958.8)
distancia_millas = haversine(4.6097, -74.0817, 6.2442, -75.5812, radio=3958.8)
print(f"Distancia en millas: {distancia_millas:.2f} mi")
```

```
Distancia en millas: 152.94 mi
```

```
# Aplicamos el "truco" matemático: si le pasamos el radio en metros, el resultado sale en metros
distancia_metros = haversine(4.6097, -74.0817, 6.2442, -75.5812, radio=6371000.0)
print(f"Distancia en metros: {distancia_metros:.2f} m")
```

```
Distancia en metros: 246135.99 m
```

```
# --- FUNCIÓN 2: PROCESADOR POR LOTES ---
def medir_ruta(lista_coords):
    # RECIBE: Una lista que contiene múltiples tuplas de coordenadas

    # Creamos una lista vacía para ir guardando las distancias calculadas
    distancias_tramos = []

    # Iniciamos un ciclo que recorre la lista. Paramos un índice antes del final (- 1)
    # para no desbordar la lista cuando intentemos buscar el punto 'i + 1'
    for i in range(len(lista_coords) - 1):

        # Extraemos latitud y longitud del punto actual (donde estamos parados)
        lat1, lon1 = lista_coords[i]

        # Extraemos latitud y longitud del punto siguiente (hacia donde vamos)
        lat2, lon2 = lista_coords[i + 1]

        # Llamamos a nuestra función 'haversine' para calcular la distancia de este tramo
        distancia_tramo = haversine(lat1, lon1, lat2, lon2)

        # Guardamos el resultado de este tramo en nuestra lista final
        distancias_tramos.append(distancia_tramo)

    # SACAMOS: La lista completa con todas las distancias de los tramos calculados
    return distancias_tramos

# Definimos una ruta con 3 puntos (2 tramos)
ruta_colombiana = [(4.6097, -74.0817), (6.2442, -75.5812), (3.4516, -76.5320)]
# Ejecutamos nuestra función procesadora
tramos = medir_ruta(ruta_colombiana)

print("\n--- Distancia por tramos en una ruta ---")
```

```
--- Distancia por tramos en una ruta ---
```

```
# Imprimimos la lista de resultados usando una comprensión de lista para redondear a 2 decimales
print(f"Tramos (km): {[round(d, 2) for d in tramos]}")
```

```
Tramos (km): [246.14, 327.9]
```

```
# --- FUNCIÓN 3: RECOLECTOR DINÁMICO ---
def describir_punto(lat, lon, **kwargs):
    # RECIBE: Latitud, longitud obligatorias, y un diccionario '**kwargs' con datos extra

    # Iniciamos creando un texto base con las coordenadas obligatorias
    descripcion = f"Punto ({lat}, {lon})"

    # Iniciamos un ciclo para "abrir" el diccionario de parámetros extra.
    # .items() nos separa el nombre de la variable (clave) y su contenido (valor)
    for clave, valor in kwargs.items():

        # Sumamos al texto base una barrita () seguida del nombre y valor del dato extra
        descripcion += f" | {clave}: {valor}"

    # SACAMOS: Un solo texto largo que concatena toda la información del punto
    return descripcion

print("\n--- Creación de atributos dinámicos ---")
```

```
--- Creación de atributos dinámicos ---
```

```
# Ejecutamos enviando variables inventadas (ciudad, elevacion, costero, clima)
print(describir_punto(4.6097, -74.0817, ciudad="Bogotá", elevacion=2640))
```

```
Punto (4.6097, -74.0817) | ciudad: Bogotá | elevacion: 2640
```

```
print(describir_punto(10.3997, -75.4795, ciudad="Cartagena", costero=True, clima="Cálido"))
```

```
Punto (10.3997, -75.4795) | ciudad: Cartagena | costero: True | clima: Cálido
```

R

```

# #| eval: false

# R no tiene una función nativa directa para radianes, la creamos rápido
pasar_a_radianes <- function(grados) {
  return(grados * pi / 180)
}

# --- FUNCIÓN 1: CÁLCULO MATEMÁTICO BÁSICO ---
haversine <- function(lat1, lon1, lat2, lon2, radio = 6371.0) {
  # RECIBE: 4 coordenadas decimales y un radio opcional (por defecto 6371.0)

  # Convierte la diferencia de latitudes a radianes llamando a nuestra mini-función
  dlat <- pasar_a_radianes(lat2 - lat1)

  # Convierte la diferencia de longitudes a radianes
  dlon <- pasar_a_radianes(lon2 - lon1)

  # Calcula 'a': Ajuste trigonométrico esférico
  a <- (sin(dlat / 2)^2 +
         cos(pasar_a_radianes(lat1)) * cos(pasar_a_radianes(lat2)) * sin(dlon / 2)^2)

  # Calcula 'c': Ángulo central en radianes usando arcotangente (atan2)
  c <- 2 * atan2(sqrt(a), sqrt(1 - a))

  # SACA: Multiplica el ángulo por el radio para darnos la distancia física
  distancia <- radio * c
  return(distancia)
}

cat("--- Cálculo de distancia simple ---\n")

```

--- Cálculo de distancia simple ---

```

# Ejecutamos la función usando el radio en kilómetros por defecto
distancia_km <- haversine(4.6097, -74.0817, 6.2442, -75.5812)
cat(sprintf("Distancia Bogotá-Medellín: %.2f km\n", distancia_km))

```

Distancia Bogotá-Medellín: 246.14 km

```
# Sobrescribimos el parámetro 'radio' para obtener el resultado en millas (3958.8)
distancia_millas <- haversine(4.6097, -74.0817, 6.2442, -75.5812, radio = 3958.8)
cat(sprintf("Distancia en millas: %.2f mi\n", distancia_millas))
```

Distancia en millas: 152.94 mi

```
# Aplicamos el "truco" matemático: si le pasamos el radio en metros, el resultado sale en metros
distancia_metros <- haversine(4.6097, -74.0817, 6.2442, -75.5812, radio = 6371000.0)
cat(sprintf("Distancia en metros: %.2f m\n", distancia_metros))
```

Distancia en metros: 246135.99 m

```
# --- FUNCIÓN 2: PROCESADOR POR LOTES ---
medir_ruta <- function(lista_coords) {
  # RECIBE: Una lista de listas numéricas (coordenadas)

  # Creamos un vector numérico vacío donde depositaremos las respuestas
  distancias_tramos <- c()

  # Iniciamos el ciclo restando 1 al tamaño total. Si no restamos 1,
  # al final del ciclo R buscará el punto 'i + 1' que no existe y arrojará error.
  for (i in 1:(length(lista_coords) - 1)) {

    # Extraemos el punto actual (i)
    punto_a <- lista_coords[[i]]

    # Extraemos el punto siguiente (i+1)
    punto_b <- lista_coords[[i + 1]]

    # Llamamos a 'haversine' dándole latitud y longitud de ambos puntos
    d <- haversine(punto_a[1], punto_a[2], punto_b[1], punto_b[2])

    # Añadimos (concatenamos) la distancia recién calculada a nuestro vector
    distancias_tramos <- c(distancias_tramos, d)
  }

  # SACAMOS: El vector repleto con las distancias de todos los tramos
  return(distancias_tramos)
}
```

```

# Definimos una ruta con 3 puntos (2 tramos)
ruta_colombiana <- list(c(4.6097, -74.0817), c(6.2442, -75.5812), c(3.4516, -76.5320))
# Ejecutamos la función
tramos <- medir_ruta(ruta_colombiana)

cat("\n--- Distancia por tramos en una ruta ---\n")

```

--- Distancia por tramos en una ruta ---

```
cat(sprintf("Tramos (km): %.2f, %.2f\n", tramos[1], tramos[2]))
```

Tramos (km): 246.14, 327.90

```

# --- FUNCIÓN 3: RECOLECTOR DINÁMICO ---
describir_punto <- function(lat, lon, ...) {
  # RECIBE: lat, lon, y unos tres puntos (...) que atrapan variables extra nombradas

  # Creamos el texto base principal
  descripcion <- sprintf("Punto (%.4f, %.4f)", lat, lon)

  # Extraemos las variables atrapadas en los '...' y las metemos a una lista
  args_extra <- list(...)

  # Verificamos si la lista tiene datos (si el usuario envió algo extra)
  if (length(args_extra) > 0) {

    # Recorremos cada nombre de variable (clave) en la lista extra
    for (clave in names(args_extra)) {

      # Pegamos (paste0) la barra, la clave y su valor al texto original
      descripcion <- paste0(descripcion, sprintf(" | %s: %s", clave, args_extra[[clave]]))
    }
  }

  # SACA: El texto unificado
  return(descripcion)
}

cat("\n--- Creación de atributos dinámicos ---\n")

```

```
--- Creación de atributos dinámicos ---
```

```
# Pasamos las coordenadas y le inventamos datos (ciudad, elevacion, etc.)  
cat(describir_punto(4.6097, -74.0817, ciudad="Bogotá", elevacion=2640), "\n")
```

```
Punto (4.6097, -74.0817) | ciudad: Bogotá | elevacion: 2640
```

```
cat(describir_punto(10.3997, -75.4795, ciudad="Cartagena", costero=TRUE), "\n")
```

```
Punto (10.3997, -75.4795) | ciudad: Cartagena | costero: TRUE
```

## Julia

```
# #| eval: false  
j_eval(r"-  
# --- FUNCIÓN 1: CÁLCULO MATEMÁTICO BÁSICO ---  
# En Julia, los argumentosopcionales con nombre DEBEN ir después de un punto y coma (;)  
function haversine(lat1, lon1, lat2, lon2; radio=6371.0)  
    # RECIBE: 4 números flotantes y un radio (por defecto en km)  
  
    # Convierte la diferencia de latitudes a radianes con la función nativa 'deg2rad'  
    dlat = deg2rad(lat2 - lat1)  
  
    # Convierte la diferencia de longitudes a radianes  
    dlon = deg2rad(lon2 - lon1)  
  
    # Calcula 'a': Ajuste esférico para la Tierra  
    a = (sin(dlat / 2)^2 +  
         cos(deg2rad(lat1)) * cos(deg2rad(lat2)) * sin(dlon / 2)^2)  
  
    # Calcula 'c': Ángulo central en radianes (Ojo: en Julia es solo 'atan', no 'atan2')  
    c = 2 * atan(sqrt(a), sqrt(1 - a))  
  
    # SACA: Multiplica el ángulo por el radio para la distancia real  
    return radio * c  
end  
  
using Printf
```

```

println("--- Cálculo de distancia simple ---")
# Ejecutamos con el radio por defecto (km)
distancia_km = haversine(4.6097, -74.0817, 6.2442, -75.5812)
@printf("Distancia Bogotá-Medellín: %.2f km\n", distancia_km)

# Sobrescribimos el parámetro 'radio' para obtener el resultado en millas (3958.8)
distancia_millas = haversine(4.6097, -74.0817, 6.2442, -75.5812, radio=3958.8)
@printf("Distancia en millas: %.2f mi\n", distancia_millas)

# Aplicamos el "truco" matemático: si le pasamos el radio en metros, el resultado sale en metros
distancia_metros = haversine(4.6097, -74.0817, 6.2442, -75.5812, radio=6371000.0)
@printf("Distancia en metros: %.2f m\n", distancia_metros)

# --- FUNCIÓN 2: PROCESADOR POR LOTES ---
function medir_ruta(lista_coords)
    # RECIBE: Un arreglo con tuplas de coordenadas

    # Creamos un arreglo vacío pero tipado (Float64) para que Julia procese más rápido
    distancias_tramos = Float64[]

    # Iniciamos el ciclo restando 1. Si no restamos 1, al final del ciclo
    # Julia intentará buscar el punto 'i + 1', el cual no existe, y colapsará.
    for i in 1:(length(lista_coords) - 1)

        # Extraemos coordenadas iniciales (i) y finales (i+1)
        lat1, lon1 = lista_coords[i]
        lat2, lon2 = lista_coords[i + 1]

        # Invocamos la función 'haversine' y 'empujamos' (push!) el dato al arreglo final
        push!(distancias_tramos, haversine(lat1, lon1, lat2, lon2))
    end

    # SACAMOS: Arreglo con las distancias parciales
    return distancias_tramos
end

# Definimos una ruta con 3 puntos (2 tramos)
ruta_colombiana = [(4.6097, -74.0817), (6.2442, -75.5812), (3.4516, -76.5320)]
# Ejecutamos la función
tramos = medir_ruta(ruta_colombiana)

```

```

println("\n--- Distancia por tramos en una ruta ---")
@printf("Tramos (km): [%.2f, %.2f]\n", tramos[1], tramos[2])

# --- FUNCIÓN 3: RECOLECTOR DINÁMICO ---
# Usamos 'kwargs...' después del punto y coma (;) para atrapar variables extra con nombre
function describir_punto(lat, lon; kwargs...)
    # RECIBE: Coordenadas y una "bolsa" con parámetros extra

    # Creamos la cadena de texto base
    descripcion = "Punto ($lat, $lon)"

    # Desempaquetamos la bolsa iterando sobre la clave (nombre de variable) y el valor
    for (clave, valor) in kwargs

        # En Julia el operador *= sirve para concatenarle más texto a una cadena existente
        descripcion *= " | $clave: $valor"
    end

    # SACA: El texto concatenado
    return descripcion
end

println("\n--- Creación de atributos dinámicos ---")
# Le mandamos variables extra (ciudad, elevacion, etc)
println(describir_punto(4.6097, -74.0817, ciudad="Bogotá", elevacion=2640))
println(describir_punto(10.3997, -75.4795, ciudad="Cartagena", costero=true))
)-

```

Starting Julia ...

### Resumen de sintaxis: estructura de funciones

Table 1: Diferencias en el empaquetado y argumentos de funciones

Característica	Python	R	Julia
Declaración	def calc(x):	calc <- function(x) { }	function calc(x) ... end
Retorno	return y	return(y)	return y

Característica	Python	R	Julia
Parámetro x defecto	<code>def calc(r=6371):</code>	<code>function(r=6371)</code>	<code>function calc(; r=6371)</code>
Múltiples args extra	<code>**kwargs</code>	<code>...</code>	<code>kwargs...</code>

## 2. Clases: organizando datos y comportamiento juntos

En el mundo del desarrollo de software, existen diferentes formas de estructurar el código. Tradicionalmente, la programación separaba los datos por un lado y las funciones que operaban sobre esos datos por el otro. Sin embargo, la **Programación Orientada a Objetos (POO)** propone un paradigma distinto: agrupar los datos y los comportamientos relacionados en una sola entidad unificada.

Para dominar este paradigma, primero debemos entender sus conceptos genéricos fundamentales:

- **Clase:** Es el plano, la plantilla o el molde abstracto. No contiene datos reales, sino que define qué estructura tendrán los datos y qué acciones se podrán realizar.
- **Objeto (o instancia):** Es la materialización de la clase. Es el producto final creado a partir del molde, que ocupa un espacio real en la memoria de la computadora y contiene datos específicos.
- **Atributos (o estado):** Son las variables internas que guardan las características únicas de cada objeto.
- **Métodos (o comportamiento):** Son las funciones que viven dentro (o asociadas) al objeto. Definen lo que el objeto “sabe hacer” con sus propios datos.
- **Constructor:** Es la puerta de entrada. Es el mecanismo inicial que toma los datos crudos y los ensambla siguiendo las reglas de la clase para dar vida a un nuevo objeto.

### Analogía entre clases/objetos y moldes/arepas

- **La Clase (el molde):** Es como el plano arquitectónico de una casa o el molde para hacer arepas. El molde define que toda arepa tendrá un grosor y un diámetro, pero el molde en sí no se puede comer.
- **El Objeto (la instancia):** Es la arepa ya hecha, salida del molde. Puedes sacar mil arepas distintas (unas de queso, otras de carne) usando el mismo molde. Todas comparten la misma estructura, pero contienen datos (rellenos) diferentes.

## El reto: de simples números a entidades inteligentes

En el análisis de datos espaciales, solemos trabajar con coordenadas crudas (simples pares de números). Si los dejamos sueltos, son solo texto sin significado. Nuestro objetivo en este ejercicio es aplicar los conceptos de la POO para tomar esos números y “darles vida”.

Queremos tomar dos simples coordenadas geográficas y convertirlas en un **Objeto Espacial Inteligente** que tenga la capacidad intrínseca de realizar tres tareas fundamentales:

1. **Representarse a sí mismo (Texto)**: Que el objeto pueda decirnos su nombre y ubicación en un formato legible para humanos.
2. **Analítica espacial (Matemáticas)**: Que el objeto tenga la capacidad de calcular la distancia hacia otro punto.
3. **Visualización (Gráficos)**: Que el objeto, al recibir un punto de destino, sepa cómo dibujar un mapa interactivo mostrando la ruta y la distancia entre ambos.

A continuación, ilustramos conceptualmente este flujo de transformación:

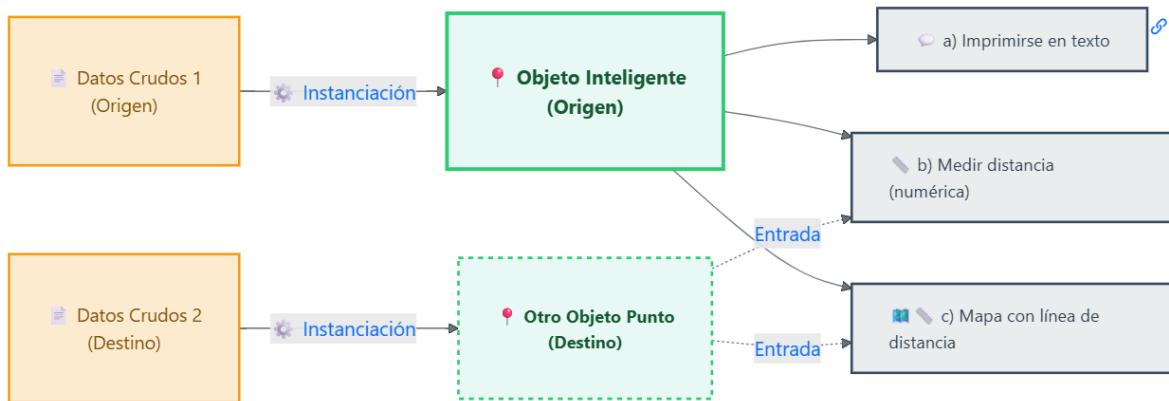


Figure 2: Flujo conceptual: Transformación de datos crudos en un objeto inteligente con capacidades.

En esta sección, crearemos una Clase llamada `PuntoEspacial`. Este “molde” le enseñará a la computadora cuatro cosas fundamentales:

## Python y la Programación Orientada a Objetos (OOP)

En Python, la creación de estructuras de datos y su comportamiento sigue un enfoque clásico y estricto de **Programación Orientada a Objetos (OOP)**. A diferencia de otros lenguajes donde los datos y las funciones operan por separado, Python prefiere encapsularlo todo (el estado y las acciones) dentro de una única fortaleza lógica: **La Clase**.

## La Analogía: El Molde y la Arepa

Para visualizar cómo Python maneja la memoria, imaginemos una cocina:

- **La Clase (class PuntoEspacial):** Es el **molde de hierro**. Por sí solo no contiene ingredientes ni alimenta a nadie; es simplemente un diseño estático que dicta la estructura geométrica, qué datos son obligatorios y qué acciones se pueden realizar.
- **El Objeto (La Instancia):** Es la **arepa física y calientita**. Cuando pasamos ingredientes reales (latitud, longitud, nombre) a través del molde, el computador reserva un espacio en memoria para crear una instancia concreta y única. Podemos sacar mil “arepas” (objetos) del mismo molde, y cada una tendrá sus propios datos independientes.

## La anatomía del molde

Cuando diseñamos una clase en Python, establecemos un contrato estricto con la máquina a través de métodos mágicos y atributos:

1. **El Constructor (`__init__`):** Es la puerta de entrada. Cada vez que intentamos crear un objeto nuevo, Python llama automáticamente a esta función para recibir los parámetros iniciales y “ensamblar” la estructura en la memoria.
2. **El Auto-reconocimiento (`self`):** En Python, la clase necesita una forma de referirse a la “arepa” específica que está cocinando en ese momento. El parámetro `self` es ese ancla; garantiza que si modificamos la latitud del “Punto A”, no estropeemos accidentalmente la latitud del “Punto B”.
3. **La representación (`__str__`):** Cómo debe presentarse el objeto si intentamos imprimirla en pantalla.
4. **Los Métodos (Comportamiento):** Son las funciones que viven **dentro** de la clase. Una vez instanciado el objeto, este carga consigo mismo todas sus herramientas. Un punto espacial en Python “sabe” cómo calcular su propia distancia hacia otro punto o cómo graficarse a sí mismo en un mapa.
  - **Los métodos analíticos (`distancia_hacia`):** Funciones que viven *dentro* del objeto. Nuestro punto será tan inteligente que sabrá calcular la distancia hacia otro punto por sí mismo, invocando internamente a la función `haversine` que programamos en la sección anterior. Además, le agregaremos un parámetro dinámico para que pueda devolver la distancia en **metros** si se lo pedimos.
  - **El método gráfico (`graficar_ruta`):** Un método que le permite al objeto usar una librería cartográfica para dibujarse a sí mismo en un mapa base. Además, usaremos nuestro método analítico para calcular el punto medio y escribir la distancia calculada directamente sobre la línea de la ruta!

A continuación, visualizamos este ciclo de vida completo: primero, el diagrama estructural del molde (la Clase) y sus métodos internos; y luego, el mapa de ejecución que muestra cómo sacamos las instancias (Objetos) a la memoria para interactuar con ellas.

## R y el sistema S3: la ilusión del molde y el poder de la etiqueta

Mientras que Python utiliza un enfoque estricto donde cada objeto nace de un “molde de hierro” predefinido, R aborda la programación orientada a objetos de una manera mucho más relajada e informal, principalmente a través de su **sistema S3**.

En R, las clases no se definen formalmente. No hay una bóveda blindada que encierre los datos y los métodos juntos. En su lugar, el sistema S3 confía en el “sistema de honor” y en el uso inteligente de etiquetas.

### La analogía: el recipiente común y el post-it

Si en Python teníamos un molde de hierro para hacer arepas, en R la cocina funciona muy distinto:

- **Los ingredientes (la lista):** En lugar de un molde, tomamos un recipiente plástico común y corriente (como el “porta” del almuerzo o una `list` en R), y echamos ahí nuestros ingredientes crudos (latitud, longitud y nombre).
- **La “clase” (el post-it):** Para convertir esa lista genérica en un objeto espacial, literalmente le pegamos un “post-it” en la tapa del recipiente. Al asignarle un atributo de clase (`class(objeto) <- 'PuntoEspacial'`), le estamos diciendo a R: *“De ahora en adelante, confía en mí y trata a este recipiente como si fuera una arepa”*.

## La anatomía del sistema S3

Dado que no hay un “molde” real, la instanciación y el comportamiento en R se dividen en tres pilares fundamentales que flotan libremente en el entorno:

1. **El constructor informal:** Es simplemente una función normal (como `nuevo_punto_espacial()`) que recibe los datos, los empaqueta en una lista y, lo más importante, le estampa la etiqueta de la clase antes de devolverla.
2. **Las funciones genéricas (los inspectores):** Funciones como `print()` o `distancia_hacia()` son genéricas. No saben cómo calcular nada por sí mismas; su único trabajo es mirar el “post-it” del objeto que reciben y decir: *“Ah, eres un PuntoEspacial, déjame buscar al experto que sabe tratar contigo”*. Esto se logra mediante el comando interno `UseMethod()`.

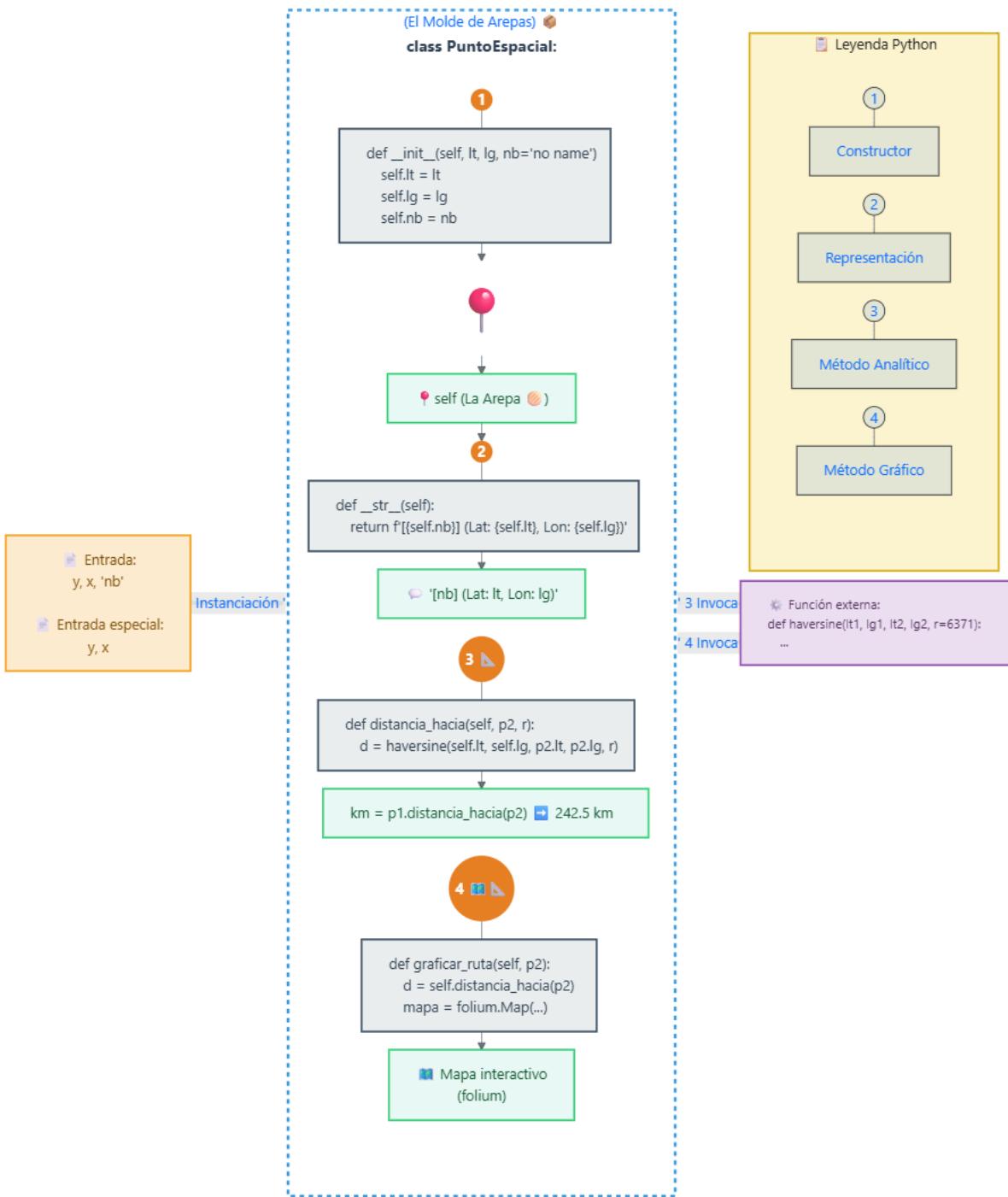


Figure 3: Diagrama del flujo de instanciación: El molde (Clase) y las arepas (Objetos).

```

# --- EJECUCIÓN DEL CÓDIGO ---

# [1 🍴] INSTANCIACIÓN: Sacando arepas del molde
punto_origen = PuntoEspacial(4.6097, -74.0817, 'Bogotá') # 🍴 Arepa 1
punto_destino = PuntoEspacial(6.2442, -75.5812, 'Medellín') # 🍴 Arepa 2
punto_misterioso = PuntoEspacial(12.5847, -81.7006) # 🍴 Arepa Especial 3

# [2 💬] REPRESENTACIÓN: Ver los objetos
print(punto_origen)
print(punto_destino)
print(punto_misterioso)

# [3 🔎] MÉTODOS ANALÍTICOS: Uso de la máquina matemática interna
resultado_km = punto_origen.distancia_hacia(punto_destino)
resultado_metros = punto_origen.distancia_hacia(punto_destino, radio=6371000.0)

# [4 📈] MÉTODO GRÁFICO: Mapas generados por el objeto
mapa_final = punto_origen.graficar_ruta(punto_destino)

```

Figure 4: Ejecución del código instanciando los objetos en memoria.

3. **Los métodos específicos (el despacho):** Son las funciones que realmente hacen el trabajo duro, y se nombran uniendo el genérico y la clase (por ejemplo, `distanzia_hacia.PuntoEspacial()`). El genérico le “despacha” el objeto a esta función especializada.

A continuación, visualizamos este paradigma en acción: primero, el diagrama arquitectónico que muestra cómo los métodos flotan fuera de la estructura de datos; y luego, el bloque de ejecución donde vemos la magia del etiquetado y el despacho S3 en tiempo real.

### Julia y el despacho múltiple: separando los ingredientes de la receta

Julia rechaza por completo el concepto tradicional de clases que vimos en Python. En su lugar, abraza un paradigma donde los datos (las variables) y el comportamiento (las funciones) están estrictamente separados, pero trabajan en una armonía perfecta gracias al núcleo absoluto del lenguaje: el **despacho múltiple** (Multiple Dispatch).

💡 La analogía: el contenedor rígido y los chefs expertos

Si Python es un molde de hierro y R es un post-it, Julia es una cocina de alta eficiencia:

- **El contenedor (struct):** Es un contenedor plástico rígido (como un “porta” de cristal). Su única función es guardar los ingredientes (latitud, longitud, nombre) de forma organizada y con etiquetas de tipo muy estrictas (`Float64`, `String`). El `struct` es “tonto”; no sabe hacer absolutamente nada por sí solo, no tiene métodos ni recetas grabadas en sus paredes.
- **El comportamiento (las funciones libres):** Son un batallón de chefs expertos que están esperando en la cocina. Cuando usted grita “¡Quiero la distancia!”, el jefe de cocina (el compilador de Julia) mira exactamente qué ingredientes le entregó en el porta. Dependiendo de los tipos de datos que usted mande, le asigna el trabajo al chef que tenga la receta matemática más optimizada para esos tipos exactos.

### La anatomía de la arquitectura en Julia

En este lenguaje, construimos la estructura definiendo tipos de datos crudos y luego escribimos funciones libres que declaran para qué combinaciones de tipos están preparadas.

1. **La estructura base (struct):** A diferencia de las clases, aquí solo organizamos las variables en la memoria. Esto garantiza un tipado fuerte que le permite a Julia ser tan rápido como C o Fortran.
2. **El constructor:** Julia crea un constructor automático básico, pero podemos crear constructores propios si necesitamos reglas especiales (como asignar ‘no name’ si el usuario olvida el nombre al llenar el contenedor).

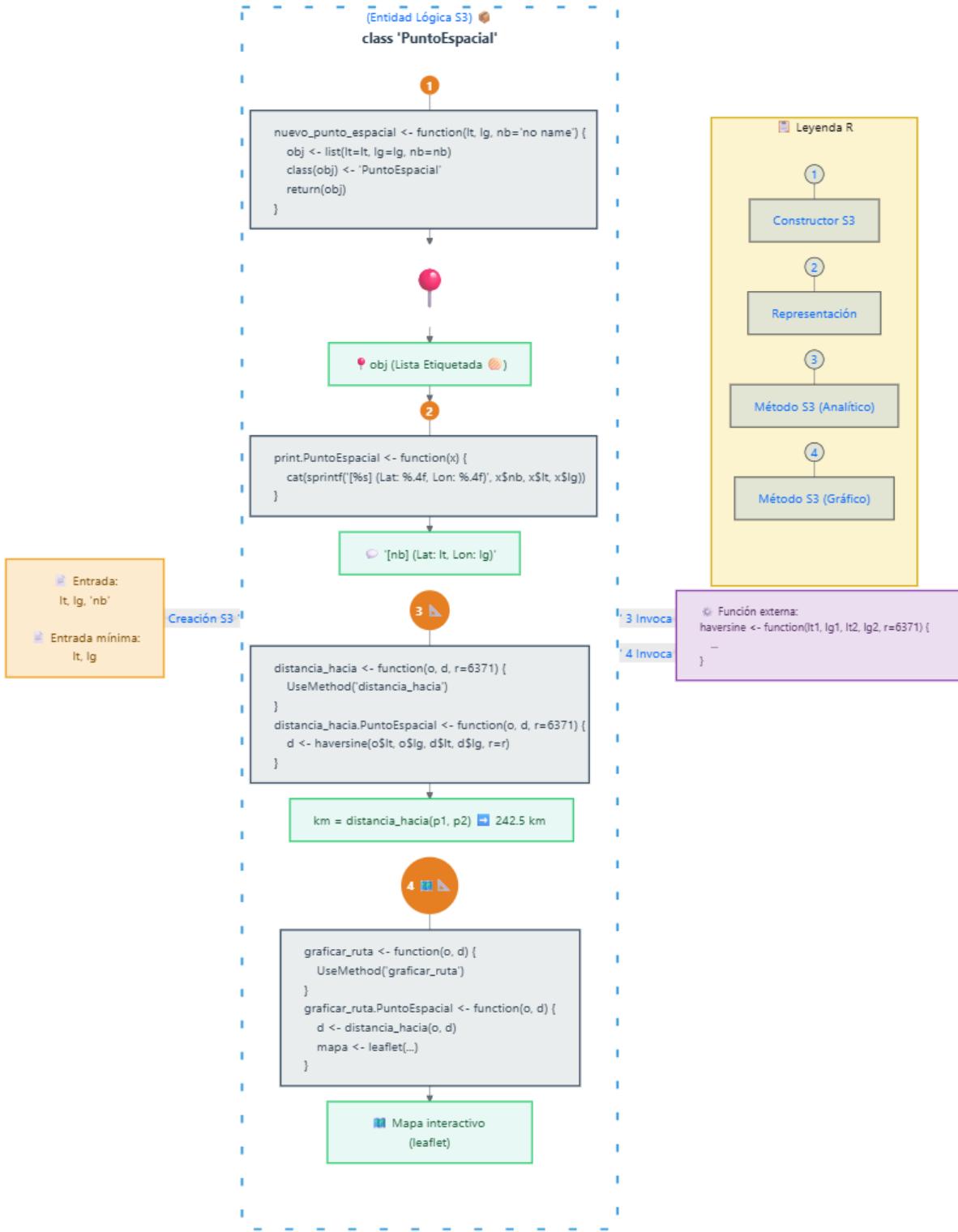


Figure 5: Diagrama del flujo de instancia en R (S3): El molde y las arepas.

```

# --- EJECUCIÓN DEL CÓDIGO EN R (S3) ---

# [1 🍴] INSTANCIACIÓN: Etiquetando listas con el molde
punto_origen = nuevo_punto_espacial(4.6097, -74.0817, 'Bogotá') # 🍴 Arepa 1
punto_destino = nuevo_punto_espacial(6.2442, -75.5812, 'Medellín') # 🍴 Arepa 2
punto_misterioso = nuevo_punto_espacial(12.5847, -81.7006) # 🍴 Arepa Especial 3

# [2 💬] REPRESENTACIÓN: R busca el método 'print.PuntoEspacial'
print(punto_origen)
print(punto_destino)
print(punto_misterioso)

# [3 🔍] MÉTODOS ANALÍTICOS: Uso del UseMethod() para despachar
resultado_km = distancia_hacia(punto_origen, punto_destino)
resultado_metros = distancia_hacia(punto_origen, punto_destino, radio = 6371000.0)

# [4 🗺️] MÉTODO GRÁFICO: Mapa interactivo leaflet
mapa_final = graficar_ruta(punto_origen, punto_destino)

```

Figure 6: Ejecución del código en R (S3) instanciando los objetos en memoria.

- 3. El despacho múltiple en acción:** La magia ocurre al llamar a una función. Si tenemos una función `distancia_hacia(a, b)`, Julia analiza el tipo de dato exacto de **todos** los argumentos que le pasamos. Basado en esa combinación de tipos, el lenguaje enruta la ejecución al método más específico de forma instantánea.

A continuación, visualizamos este potente paradigma: primero, la representación de cómo el `struct` y el despacho múltiple interactúan conceptualmente en una “caja virtual” (con líneas muy espaciadas para notar que no es una prisión de código); y luego, el bloque de ejecución donde vemos a Julia activando los métodos correctos al vuelo.

### Laboratorio comparativo: implementación del código completo

A continuación, presentamos el código completo y funcional para los tres lenguajes. Te invitamos a navegar entre las pestañas para comparar cómo cada arquitectura resuelve el mismo problema.

**Tip de lectura:** Busca las etiquetas con emojis ([1 ], [2 ], [3 ], [4 ]) dentro de los comentarios del código. Estas te ayudarán a conectar cada bloque de código directamente con los diagramas de arquitectura que vimos en las secciones anteriores.

### Python

```
# #| echo: false
# #| eval: false
import math
import folium # Librería indispensable en Python para generar mapas interactivos

# --- 0. TRAEMOS NUESTRA MÁQUINA MATEMÁTICA ---
# Pegamos aquí la función que ya habíamos creado para que este código funcione por sí solo
def haversine(lat1, lon1, lat2, lon2, radio=6371.0):
    dlat, dlon = math.radians(lat2 - lat1), math.radians(lon2 - lon1)
    a = math.sin(dlat / 2)**2 + math.cos(math.radians(lat1)) * math.cos(math.radians(lat2)) * math.sin(dlon / 2)**2
    return radio * (2 * math.atan2(math.sqrt(a), math.sqrt(1 - a)))

# --- 1. DEFINICIÓN DEL MOLDE (LA CLASE) ---
# Usamos la palabra reservada 'class' seguida del nombre (en mayúscula por convención)
class PuntoEspacial:

    # [1 ] 1.1 EL CONSTRUCTOR (__init__)
    # Esta función especial se ejecuta automáticamente el momento en que "nace" un punto.
    # 'self' significa "yo mismo". Es la forma en que el objeto se refiere a sus propios datos
```

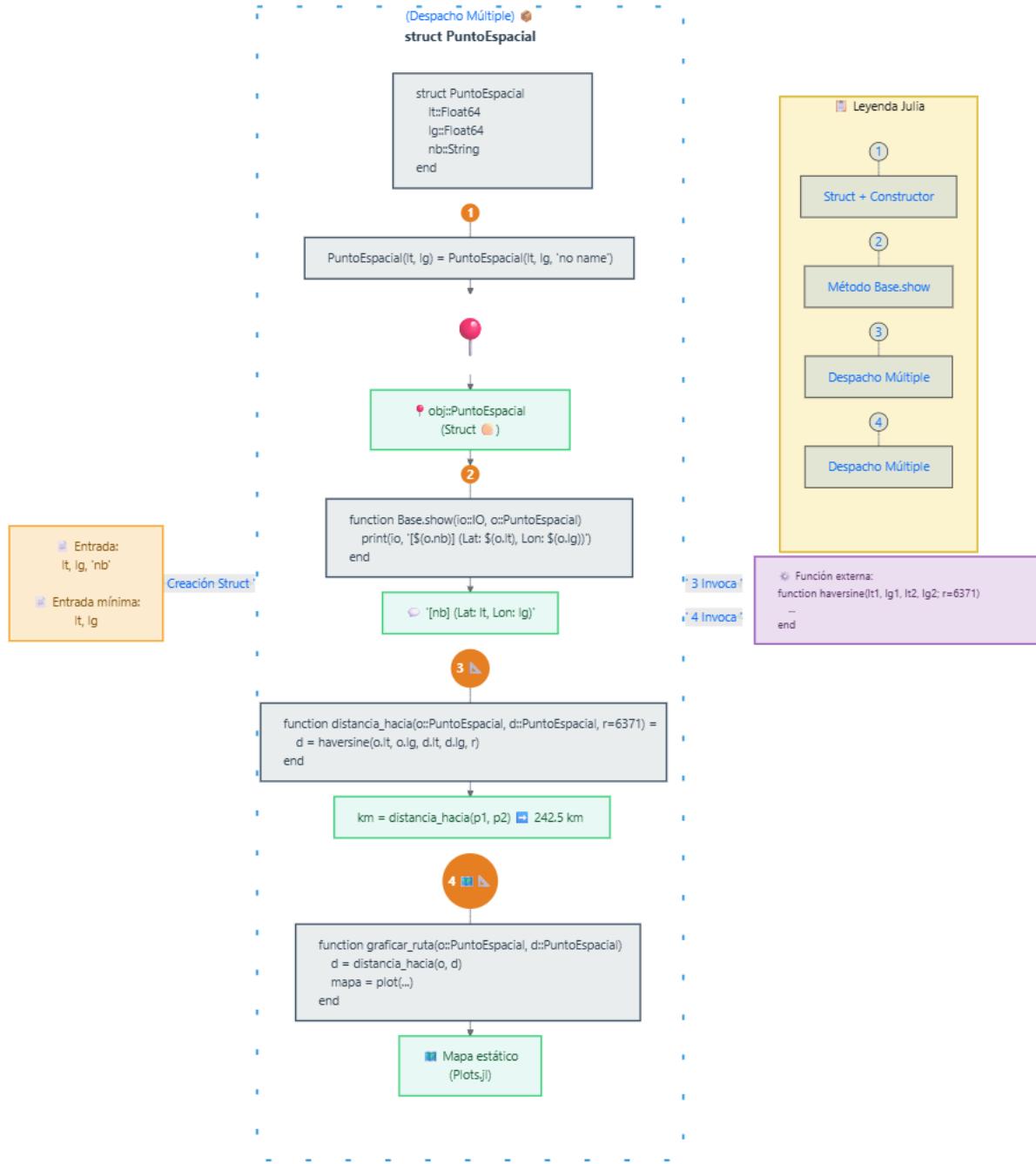


Figure 7: Diagrama del flujo de instantiación en Julia (Despacho Múltiple): El molde (Struct) y las arepas.

```

# --- EJECUCIÓN DEL CÓDIGO EN JULIA ---

# [1 🍴] INSTANCIACIÓN: Construyendo la estructura (struct)
punto_origen = PuntoEspacial(4.6097, -74.0817, 'Bogotá') # 🍴 Arepa 1
punto_destino = PuntoEspacial(6.2442, -75.5812, 'Medellín') # 🍴 Arepa 2
punto_misterioso = PuntoEspacial(12.5847, -81.7006, '') # 🍴 Arepa Especial 3

# [2 💬] REPRESENTACIÓN: Julia activa automáticamente Base.show()
println(punto_origen)
println(punto_destino)
println(punto_misterioso)

# [3 🔎] MÉTODOS ANALÍTICOS: El Despacho Múltiple elige el método correcto
resultado_km = distancia_hacia(punto_origen, punto_destino)
resultado_metros = distancia_hacia(punto_origen, punto_destino, radio = 6371000.0)

# [4 🗺️ 🔎] MÉTODO GRÁFICO: Ejecutando la función tipada
mapa_final = graficar_ruta(punto_origen, punto_destino)

```

Figure 8: Ejecución del código en Julia instanciando las estructuras en memoria.

```

# Definimos que latitud y longitud son obligatorios, pero el nombre tiene un valor por defecto
def __init__(self, latitud, longitud, nombre="Punto sin nombre"):

    # Guardamos la latitud que nos entregaron dentro del cuerpo del objeto (self)
    self.latitud = latitud

    # Guardamos la longitud dentro del objeto
    self.longitud = longitud

    # Guardamos el nombre dentro del objeto
    self.nombre = nombre

# [2 ] 1.2 LA REPRESENTACIÓN TEXTUAL (__str__)
# Si le decimos a Python print(objeto), por defecto imprimirá basura técnica.
# Esta función especial le enseña a Python cómo queremos que se lea el objeto en pantalla.
def __str__(self):
    # SACA: Un texto bonito combinando el nombre y las coordenadas
    return f"[{self.nombre}] (Lat: {self.latitud}, Lon: {self.longitud})"

# [3 ] 1.3 LOS MÉTODOS (COMPORTAMIENTO ANALÍTICO)
# Es una función normal, pero como tiene 'self', vive dentro del objeto.
# Agregamos el parámetro 'radio' para que el usuario pueda pedir el resultado en metros.
def distancia_hacia(self, otro_punto, radio=6371.0):

    # Llamamos a nuestra función haversine externa pasándole el radio solicitado
    # Le enviamos NUESTRAS coordenadas (self) y las coordenadas DEL OTRO (otro_punto)
    distancia = haversine(self.latitud, self.longitud, otro_punto.latitud, otro_punto.longitud)

    # SACA: La distancia numérica calculada entre los dos objetos
    return distancia

# [4 ] 1.4 EL MÉTODO GRÁFICO (MAPA BASE CON ETIQUETA VISIBLE)
# Este método le da la habilidad al objeto de dibujarse en un mapa web interactivo.
def graficar_ruta(self, otro_punto):

    # Primero, le decimos al objeto que calcule su propia distancia hacia el destino
    distancia_calculada = self.distancia_hacia(otro_punto)

    # Calculamos el punto medio geométrico para centrar la cámara y poner el texto
    centro_lat = (self.latitud + otro_punto.latitud) / 2
    centro_lon = (self.longitud + otro_punto.longitud) / 2

```

```

# Creamos el lienzo del mapa base (estilo OpenStreetMap por defecto)
mapa = folium.Map(location=[centro_lat, centro_lon], zoom_start=6)

# Ponemos un pin en nuestra ubicación de origen (self)
folium.Marker([self.latitud, self.longitud], popup=self.nombre).add_to(mapa)

# Ponemos un pin en el destino (otro_punto)
folium.Marker([otro_punto.latitud, otro_punto.longitud], popup=otro_punto.nombre).add_to(mapa)

# Trazamos una línea roja que conecte ambas coordenadas
folium.PolyLine([(self.latitud, self.longitud), (otro_punto.latitud, otro_punto.longitud)]).add_to(mapa)

# ¡NUEVO!: Ponemos un marcador transparente en el medio con un 'Tooltip' permanente.
# Esto obliga a Folium a mostrar el texto de la distancia sin necesidad de hacer clic
etiqueta = f"Distancia: {distancia_calculada:.2f} km"
folium.Marker(
    [centro_lat, centro_lon],
    icon=folium.DivIcon(html=""),
    # Ocultamos el ícono tradicional
    tooltip=folium.Tooltip(etiqueta, permanent=True, direction="top")
).add_to(mapa)

# SACA: El objeto mapa listo para ser mostrado
return mapa

# --- 2. CREACIÓN DE OBJETOS (INSTANCIACIÓN) ---

print("--- Creando objetos a partir del molde ---")

--- Creando objetos a partir del molde ---

# [1] INSTANCIACIÓN: Creamos el primer objeto (Bogotá).
# Noten que no le pasamos el parámetro 'self', Python hace eso invisiblemente.
punto_origen = PuntoEspacial(4.6097, -74.0817, "Bogotá")

# Creamos el segundo objeto (Medellín). Usamos los mismos planos, pero con otros datos.
punto_destino = PuntoEspacial(6.2442, -75.5812, "Medellín")

# Creamos un tercer objeto, pero omitimos el nombre para ver si funciona el valor por defecto
punto_misterioso = PuntoEspacial(12.5847, -81.7006)

```

```
# [2] REPRESENTACIÓN: Probamos nuestra función especial __str__ imprimiendo los objetos
print(punto_origen)
```

[Bogotá] (Lat: 4.6097, Lon: -74.0817)

```
print(punto_destino)
```

[Medellín] (Lat: 6.2442, Lon: -75.5812)

```
print(punto_misterioso)
```

[Punto sin nombre] (Lat: 12.5847, Lon: -81.7006)

```
print("\n--- Usando los métodos del objeto ---")
```

--- Usando los métodos del objeto ---

```
# [3] MÉTODOS ANALÍTICOS: Le pedimos al objeto que mida la distancia hacia 'punto_destino'
resultado_km = punto_origen.distancia_hacia(punto_destino)
print(f"La distancia desde {punto_origen.nombre} hasta {punto_destino.nombre} es: {resultado_km} km")
```

La distancia desde Bogotá hasta Medellín es: 246.14 km

```
# Truco de conversión: Pedimos el resultado en metros modificando el parámetro 'radio' a 6371000.0
resultado_metros = punto_origen.distancia_hacia(punto_destino, radio=6371000.0)
print(f"O expresado en metros: {resultado_metros:.2f} m")
```

O expresado en metros: 246135.99 m

```
# [4] MÉTODO GRÁFICO: Finalmente, le pedimos al objeto que dibuje el mapa de su ruta
mapa_final = punto_origen.graficar_ruta(punto_destino)
print("\n;Mapa interactivo generado con éxito en la variable 'mapa_final'!")
```

;Mapa interactivo generado con éxito en la variable 'mapa\_final'!

```
mapa_final
```

```
<folium.folium.Map object at 0x7faf0f39eae0>
```

R

```
# #| eval: false
# #| echo: false
# Cargamos leaflet, la librería top para mapas base interactivos en R
library(leaflet)

# R funciona diferente a Python. Usa un sistema llamado "S3" donde construimos
# listas normales y les pegamos una "etiqueta" invisible para volverlas objetos.

# --- 0. TRAEMOS NUESTRA MÁQUINA MATEMÁTICA ---
haversine <- function(lat1, lon1, lat2, lon2, radio = 6371.0) {
  rad <- function(g) { g * pi / 180 }
  a <- sin(rad(lat2 - lat1) / 2)^2 + cos(rad(lat1)) * cos(rad(lat2)) * sin(rad(lon2 - lon1))
  return(radio * (2 * atan2(sqrt(a), sqrt(1 - a))))
}

# --- 1. DEFINICIÓN DEL MOLDE (FUNCIÓN CONSTRUCTORA) ---
# [1] 1.1 EL CONSTRUCTOR INFORMAL
# En R, el "molde" es simplemente una función que empaca los datos
nuevo_punto_espacial <- function(latitud, longitud, nombre = "Punto sin nombre") {

  # Creamos una lista tradicional de R para guardar los atributos
  objeto <- list(
    latitud = latitud,
    longitud = longitud,
    nombre = nombre
  )

  # AQUÍ OCURRE LA MAGIA: Le pegamos la etiqueta de Clase "PuntoEspacial" a nuestra lista
  class(objeto) <- "PuntoEspacial"

  # SACA: El objeto ya etiquetado y listo para usarse
  return(objeto)
}
```

```

# [2 ] 1.2 LA REPRESENTACIÓN TEXTUAL (Sobrescribir 'print')
# Le enseñamos a la función nativa print() de R qué hacer cuando reciba un "PuntoEspacial"
print.PuntoEspacial <- function(obj) {
  # Imprimimos un texto bonito en pantalla usando cat() y sprintf()
  cat(sprintf("[%s] (Lat: %.4f, Lon: %.4f)\n", obj$nombre, obj$latitud, obj$longitud))
}

# [3 ] 1.3 LOS MÉTODOS (COMPORTAMIENTO ANALÍTICO)
# Primero creamos una "función genérica" vacía que avisa que este método existe,
# y le agregamos el parámetro opcional 'radio' para poder convertir a metros.
distancia_hacia <- function(obj_origen, obj_destino, radio = 6371.0) {
  UseMethod("distancia_hacia")
}

# Luego le decimos a R cómo debe funcionar ese método SOLO para la clase "PuntoEspacial"
distancia_hacia.PuntoEspacial <- function(obj_origen, obj_destino, radio = 6371.0) {

  # Extraemos los datos de ambos objetos y llamamos a nuestra máquina matemática
  distancia <- haversine(obj_origen$latitud, obj_origen$longitud,
                         obj_destino$latitud, obj_destino$longitud, radio = radio)

  # SACÁ: La distancia calculada
  return(distancia)
}

# [4 ] 1.4 EL MÉTODO GRÁFICO (MAPA BASE CON ETIQUETA)
# Declaramos la función genérica
graficar_ruta <- function(obj_origen, obj_destino) {
  UseMethod("graficar_ruta")
}

# Creamos el comportamiento específico para nuestros objetos
graficar_ruta.PuntoEspacial <- function(obj_origen, obj_destino) {

  # Calculamos la distancia usando nuestro propio método analítico
  distancia_calc <- distancia_hacia(obj_origen, obj_destino)

  # Calculamos el punto medio de la ruta para anclar el texto de la distancia
  mid_lat <- (obj_origen$latitud + obj_destino$latitud) / 2
  mid_lon <- (obj_origen$longitud + obj_destino$longitud) / 2

  # Creamos el lienzo, añadimos baldosas (OpenStreetMap) y los marcadores/líneas
}

```

```

mapa <- leaflet() %>%
  addTiles() %>% # Añade el mapa base tradicional
  addMarkers(lng = c(obj_origen$longitud, obj_destino$longitud),
             lat = c(obj_origen$latitud, obj_destino$latitud),
             popup = c(obj_origen$nombre, obj_destino$nombre)) %>%
  addPolylines(lng = c(obj_origen$longitud, obj_destino$longitud),
               lat = c(obj_origen$latitud, obj_destino$latitud),
               color = "red", weight = 3) %>%
  # ¡NUEVO!: Clavamos un popup fijo en el centro geométrico con el resultado
  addPopups(lng = mid_lon, lat = mid_lat,
             popup = sprintf("<b>Distancia:</b> %.2f km", distancia_calc))

# SACA: El mapa interactivo listo
return(mapa)
}

# --- 2. CREACIÓN DE OBJETOS (INSTANCIACIÓN) ---

cat("--- Creando objetos a partir del molde ---\n")

```

--- Creando objetos a partir del molde ---

```

# [1] INSTANCIACIÓN: Usamos nuestra función constructora para crear a Bogotá
punto_origen <- nuevo_punto_espacial(4.6097, -74.0817, "Bogotá")

# Creamos a Medellín
punto_destino <- nuevo_punto_espacial(6.2442, -75.5812, "Medellín")

# Creamos un punto sin nombre para probar el valor por defecto
punto_misterioso <- nuevo_punto_espacial(12.5847, -81.7006)

# [2] REPRESENTACIÓN: Probamos nuestra modificación al comando print()
print(punto_origen)

```

[Bogotá] (Lat: 4.6097, Lon: -74.0817)

```
print(punto_destino)
```

[Medellín] (Lat: 6.2442, Lon: -75.5812)

```
print(punto_misterioso)
```

[Punto sin nombre] (Lat: 12.5847, Lon: -81.7006)

```
cat("\n--- Usando los métodos del objeto ---\n")
```

--- Usando los métodos del objeto ---

```
# [3 ] MÉTODOS ANALÍTICOS: Ejecutamos nuestra función de clase enviando los dos objetos
resultado_km <- distancia_hacia(punto_origen, punto_destino)
cat(sprintf("La distancia desde %s hasta %s es: %.2f km\n",
            punto_origen$nombre, punto_destino$nombre, resultado_km))
```

La distancia desde Bogotá hasta Medellín es: 246.14 km

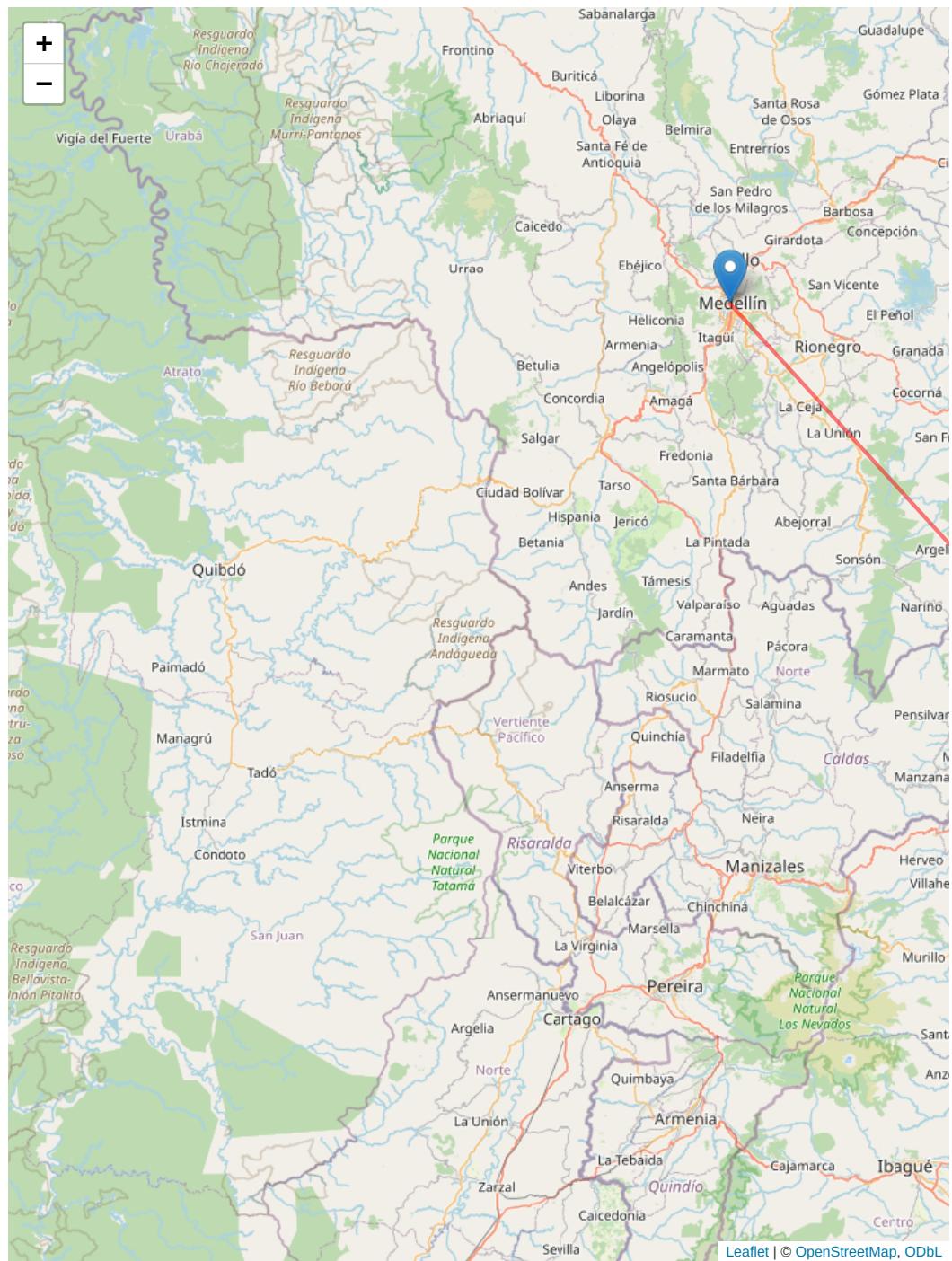
```
# Truco de conversión: Forzamos que el método devuelva la respuesta en metros
resultado_metros <- distancia_hacia(punto_origen, punto_destino, radio = 6371000.0)
cat(sprintf("O expresado en metros: %.2f m\n", resultado_metros))
```

O expresado en metros: 246135.99 m

```
# [4 ] MÉTODO GRÁFICO: Lanzamos la creación del mapa
mapa_final <- graficar_ruta(punto_origen, punto_destino)
cat("\n;Mapa interactivo generado con éxito en la variable 'mapa_final'!\n")
```

;Mapa interactivo generado con éxito en la variable 'mapa\_final'!

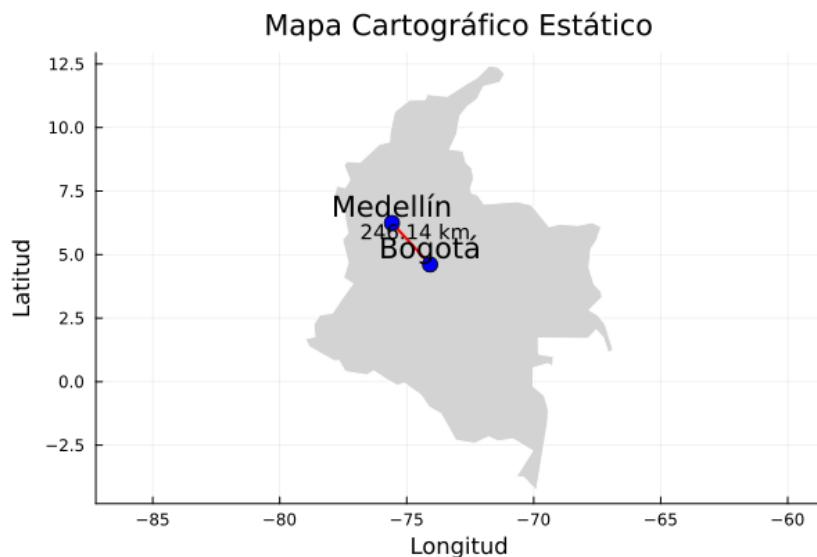
```
mapa_final
```



## Julia

### Nota Técnica: El Motor de Interoperabilidad Geomática

Si revisas el código fuente de este documento Quarto, notarás que en **Julia** estamos usando dos funciones especiales llamadas `j_eval()` y `j_plot()`, en lugar del bloque de código estándar. Esto se debe a que estamos usando un Motor de Interoperabilidad que conecta R y Julia por debajo. Para garantizar que la memoria se mantenga y el gráfico se exporte correctamente en el PDF/HTML, usamos `j_eval()` para compilar la lógica, y `j_plot()` **únicamente** al final para pedirle a la librería de gráficos que procese la imagen.



## Resumen de sintaxis: programación orientada a objetos

Como pudiste observar, cada lenguaje aborda las clases con una filosofía arquitectónica distinta: Python encapsula los métodos dentro del objeto, R usa funciones sueltas que leen etiquetas invisibles (S3), y Julia usa estructuras rígidas con funciones que exigen el tipo de dato exacto (despacho múltiple).

Table 2: Diferencias en la creación de Clases y Objetos

Característica	Python (POO Clásica)	R (Sistema S3)	Julia (Despacho Múltiple)
<b>Definir Molde</b>	<code>class Objeto:</code>	Función que retorna una <code>list()</code>	<code>struct Objeto ...</code> <code>end</code>
<b>Constructor</b>	<code>def __init__(self):</code>	Asignar etiqueta: <code>class(obj) &lt;- 'Tipo'</code>	<code>Objeto() (Automático o explícito)</code>
<b>Auto-referencia</b>	Usa la palabra <code>self</code>	Pasa el objeto <code>obj</code> como argumento	Pasa el objeto tipado <code>obj::Tipo</code>
<b>Métodos</b>	Van dentro de <code>class</code>	Genérico ( <code>UseMethod</code> ) + <code>metodo.Clase</code>	Funciones independientes enrutadas por tipo
<b>Representación</b>	<code>def __str__(self):</code>	<code>print.Clase &lt;- function(obj)</code>	<code>Base.show(io, obj::Clase)</code>

## Guías de diseño para funciones y clases

Escribir código que la máquina entienda es fácil; escribir código que otro ser humano (o usted mismo en seis meses) pueda entender, es un arte. Ya sea que esté usando el molde estricto de Python, las etiquetas de R o los chefs de Julia, existen reglas de oro universales para que su arquitectura sea sólida, limpia y profesional.

A continuación, presentamos los mandamientos del buen diseño de software para la ciencia de datos.

### 1. El diseño de Funciones: los obreros de la obra

Las funciones son la unidad básica de acción en su código. Si sus funciones son un enredo, todo el proyecto lo será.

- **Hagan una sola cosa (Single Responsibility):** Una función debe tener un único propósito claro. Si su función descarga los datos, los limpia, calcula la distancia y de paso le imprime un mapa, está haciendo demasiado. Divídala en cuatro funciones pequeñas. La navaja suiza es útil para acampar, pero en programación es un peligro.
- **Nombres que cuenten la historia:** El nombre de una función debe ser un verbo que describa exactamente su acción. Evite abreviaturas jeroglíficas. Es preferible escribir `calcular_distancia_km()` que `calc_dist()`.

- **Evite las “novelas” (Mantenlas cortas):** Si el cuerpo de su función no cabe en la pantalla de su monitor sin hacer *scroll*, es momento de machetearla y dividirla. Las funciones cortas son fáciles de testear y rara vez esconden errores.
- **Sin efectos secundarios (Pureza):** Lo ideal es que una función reciba datos, haga su trabajo y devuelva un resultado sin alterar silenciosamente variables globales u otras partes del sistema. Lo que pasa en la función, se queda en la función.

 **La regla DRY (Don't Repeat Yourself)**

“No te repitas”. Si usted se descubre copiando y pegando el mismo bloque de código tres veces seguidas y solo cambiándole una variable, deténgase. Ese bloque debe convertirse inmediatamente en una función.

---

## 2. El diseño de Clases (y Estructuras): los cimientos del edificio

Cuando decida agrupar sus datos y funciones en un objeto o estructura, hágalo con prudencia. No todo necesita ser una clase.

- **Alta cohesión:** Todo lo que viva dentro de su clase (o **struct**) debe estar íntimamente relacionado. Siguiendo nuestra analogía, no meta la receta del sancocho en el molde de las arepas. Si un **PuntoEspacial** tiene latitud y longitud, perfecto; pero si también le agrega un atributo para el “color\_favorito\_del\_usuario”, está rompiendo la cohesión.
- **Oculte las tuberías (Encapsulamiento):** El usuario de su objeto no necesita saber cómo funcionan las matemáticas internamente, solo necesita el resultado. Exponga únicamente los métodos y atributos que son estrictamente necesarios para usar el objeto (la interfaz pública) y mantenga la complejidad “escondida” en el motor.
- **Evite el “Síndrome de Dios”:** Una “clase Dios” es aquella que sabe demasiado y hace de todo (ej. **GestorDeMapasYDatosYCalculos**). Si su clase maneja toda la lógica del proyecto, divídala en objetos más pequeños y delegue responsabilidades. Un objeto **Coordenada** interactúa con un objeto **Mapa**, no son el mismo ente.
- **Si no hay estado, no hay clase:** Especialmente en Python y R. Si usted crea una clase que solo tiene métodos pero no guarda ningún dato (atributos), no necesita una clase; lo que usted necesita es un simple archivo con funciones sueltas.

## 3. Comentarios y documentación

- **El código explica el *Qué*, el comentario explica el *Por qué*:** No escriba comentarios obvios como `# Esto suma 2 + 2`. Use los comentarios para explicar decisiones de

negocio o atajos matemáticos: `# Usamos la fórmula de Haversine porque es más precisa para distancias cortas.`

- **Docstrings siempre:** Todo molde (clase/struct) y toda función importante debe tener una breve cadena de documentación en la parte superior explicando qué recibe (argumentos) y qué devuelve (retorno).

## Resumen de aprendizajes (cheat sheet)

En este capítulo hemos explorado cómo tomar decisiones lógicas y automatizar tareas repetitivas. A continuación, se presenta tu **Hoja de Referencia (Cheat Sheet)** para traducir estos conceptos estructurales entre Python, R y Julia.

### 1. ...

Table 3: Equi.. 5,25,25,25]“}

Estructura / Operador	Python	R	Julia
-----------------------	--------	---	-------

## Ejercicios

Para poner en práctica los conceptos aprendidos sobre bucles y declaraciones condicionales, deberás resolver los siguientes dos ejercicios. Puedes elegir resolverlos en **Python, R o Julia** (o implementar la solución en varios lenguajes si deseas retarte).

### Ejercicio 1: Clasificación iterativa (ciclo for e if-else)

...

### Ejercicio 2: Monitoreo de sensores (ciclo while y control de flujo)

...

## **Entregables y criterios de evaluación**

El objetivo de esta evaluación no es solo que el código funcione, sino que seas capaz de documentar y explicar tus decisiones técnicas de control de flujo.

**1. Archivos de Código:** Debes desarrollar los algoritmos en al menos uno de los siguientes formatos de archivo:

- Script tradicional (.py, .R, .jl)
- Notebook interactivo (.ipynb)
- Documento computacional (.qmd con *chunks* de código)

**2. Documento Analítico (Quarto):** Independientemente del formato de tu código fuente, debes redactar un documento en Quarto (.qmd) y renderizarlo tanto en HTML como en PDF. En este documento debes incluir tus bloques de código y responder argumentativamente a las siguientes preguntas:

- **Sobre el Ejercicio 1 (Indentación y Bloques):** Si estuvieras desarrollando el código en Python, ¿cómo sabe el computador qué instrucciones pertenecen dentro del ciclo `for`, cuáles pertenecen al `if`, y cuáles deben ejecutarse solo al final del programa? ¿Qué diferencia estructural existe frente a cómo lo resolverías en R o Julia?
- **Sobre el Ejercicio 2 (Ciclos Infinitos):** En el ciclo `while` de la simulación del río, ¿qué sucedería exactamente con la memoria y la ejecución del programa si olvidas incluir la instrucción que incrementa la variable `lectura_actual` en 1? ¿Por qué este riesgo no existe al usar un ciclo `for`?
- **Pregunta General (Operadores Lógicos):** Si estás programando en R o Julia, explica detalladamente por qué es peligroso usar el operador doble (`&&`) cuando intentas filtrar un vector completo de altitudes, y por qué en ese caso específico debes usar el operador sencillo (`&` o `.&`).

**3. Repositorio en GitHub:** Sube tu carpeta del proyecto (que debe contener tus scripts, el archivo .qmd y los renders finales en HTML y PDF) a un repositorio público en tu cuenta personal de GitHub.

- **Entrega:** Deberás enviar únicamente el enlace (URL) a tu repositorio de GitHub para la calificación.