MASTER EN BIOINFORMÁTICA Y DIGITAL HEALTH
(MBIOINDI)

TEMA 8: Introducción a la programación

12/12/2024

Miguel Rodríguez PhD
Ingeniero Bioinformático
miguel@avatarcognition.com

CV PONENTE

EDUCACIÓN

- Grado en Genética (UAB)
- Master en Bioinformática (UAB)
- Doctorado en Biomedicina (UPF)

EXPERIENCIA LABORAL

- Centro de regulación Genómica (CRG)
- Wellcome Trust Sanger Institute
- Avatar Cognition (Posición Actual)

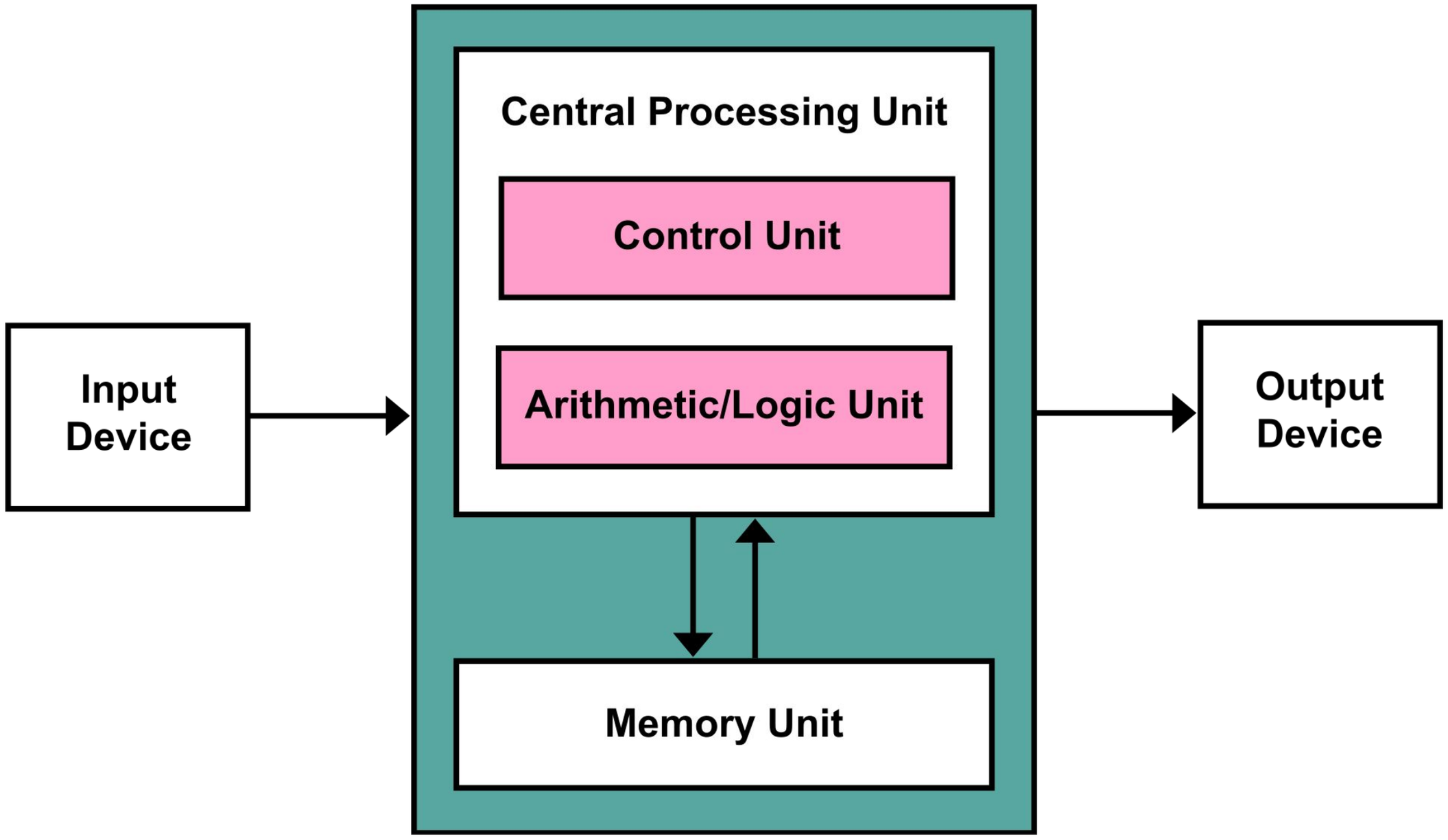
DOCENCIA

- Grado en Biología Humana (UPF)
- Master en análisis de datos ómicos (UVic)

CONTENIDOS

- Introducción a conceptos básicos de programación
- Diferencias entre paradigmas de programación
- Conceptos esenciales de variables y tipos de datos
- Instalación y descripción de software
- Introducción a la visualización de datos

¿Qué es la Programación?



¿Qué es la Programación?

¿Qué es la Programación?

Problem

A **string** is simply an ordered collection of symbols selected from some **alphabet** and formed into a word; the **length** of a string is the number of symbols that it contains.

An example of a length 21 **DNA string** (whose alphabet contains the symbols 'A', 'C', 'G', and 'T') is "ATGCTTCAGAAAGGTCTTACG."

Given: A DNA string s of length at most 1000 nt.

Return: Four integers (separated by spaces) counting the respective number of times that the symbols 'A', 'C', 'G', and 'T' occur in s.

Sample Dataset

```
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
```

Sample Output

```
20 12 17 21
```

¿Qué es la Programación?

```python
# Definimos una función para contar los nucleótidos en una cadena de ADN
def contar_nucleotidos(secuencia):
    """
    Cuenta la cantidad de nucleótidos A, C, G y T en una secuencia de ADN.

    Parámetros:
    secuencia (str): Una cadena de ADN que contiene los caracteres 'A', 'C', 'G' y 'T'.

    Retorno:
    tuple: Una tupla con cuatro valores enteros, que representan el conteo de 'A', 'C', 'G' y 'T' respectivamente.
    """
    # Contamos cada nucleótido usando el método count() de las cadenas de texto
    conteo_A = secuencia.count('A')
    conteo_C = secuencia.count('C')
    conteo_G = secuencia.count('G')
    conteo_T = secuencia.count('T')

    # Devolvemos los conteos como una tupla de cuatro enteros
    return conteo_A, conteo_C, conteo_G, conteo_T

# Ejemplo de uso de la función con la cadena de muestra
secuencia = "AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC"
resultado = contar_nucleotidos(secuencia)

# Mostramos el resultado en el formato solicitado (con valores separados por espacios)
print(" ".join(map(str, resultado)))
```

Paradigmas de programación

Paradigma Orientado a Objetos (POO)

- La **Programación Orientada a Objetos** busca representar conceptos o elementos del mundo real (o de un problema específico) como **objetos** en el código.
- Estos **objetos** son "instancias" o ejemplos concretos de **clases**, que son estructuras generales que definen las características y el comportamiento de dichos objetos.
- Cada **clase** actúa como un molde que define las propiedades (**atributos**) y comportamientos (**métodos**) de los objetos.
- Este paradigma permite modelar problemas del mundo real, haciéndolo ideal para sistemas complejos.

Paradigmas de programación

```python
class Secuencia:    # Definimos la clase Secuencia
    def __init__(self, cadena):
        """Inicializa la clase Secuencia con una cadena de ADN y calcula los conteos como atributos."""
        self.cadena = cadena
        self.conteo_A = self.cadena.count('A'); self.conteo_C = self.cadena.count('C')
        self.conteo_G = self.cadena.count('G'); self.conteo_T = self.cadena.count('T')

    def describir(self):
        """Devuelve una descripción de la secuencia con los conteos de nucleótidos."""
        return (f"Secuencia: {self.cadena}\n"
                f"Conteo de A: {self.conteo_A}\t Conteo de C: {self.conteo_C}\n"
                f"Conteo de G: {self.conteo_G}\t Conteo de T: {self.conteo_T}")
```

Paradigmas de programación

```python
class Secuencia:    # Definimos la clase Secuencia
    def __init__(self, cadena):
        """Inicializa la clase Secuencia con una cadena de ADN y calcula los conteos como atributos."""
        self.cadena = cadena
        self.conteo_A = self.cadena.count('A'); self.conteo_C = self.cadena.count('C')
        self.conteo_G = self.cadena.count('G'); self.conteo_T = self.cadena.count('T')

    def describir(self):
        """Devuelve una descripción de la secuencia con los conteos de nucleótidos."""
        return (f"Secuencia: {self.cadena}\n"
                f"Conteo de A: {self.conteo_A}\t Conteo de C: {self.conteo_C}\n"
                f"Conteo de G: {self.conteo_G}\t Conteo de T: {self.conteo_T}")
```

```python
secuencia = Secuencia("AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC")
print(secuencia.describir())
```

✓ 0.0s

```
Secuencia: AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC
Conteo de A: 20 Conteo de C: 12
Conteo de G: 17 Conteo de T: 21
```

Paradigmas de programación

```python
class Secuencia:    # Definimos la clase Secuencia
    def __init__(self, cadena):
        """Inicializa la clase Secuencia con una cadena de ADN y calcula los conteos como atributos."""
        self.cadena = cadena; self.reversa_complementaria = self.calcular_reversa_complementaria()
        self.conteo_A = self.cadena.count('A'); self.conteo_C = self.cadena.count('C')
        self.conteo_G = self.cadena.count('G'); self.conteo_T = self.cadena.count('T')

    # def describir(self):

    def calcular_reversa_complementaria(self):
        """Calcula y devuelve la reversa complementaria de la cadena de ADN."""
        complementos = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}  # Mapa de complementos
        complementaria = "".join(complementos[base] for base in self.cadena)  # Genera la complementaria
        return complementaria[::-1]  # Revierte la cadena complementaria
```

✓ 0.0s

```python
# Ejemplo de uso de la clase Secuencia
secuencia = Secuencia("AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGC")
print(secuencia.calcular_reversa_complementaria())
```

✓ 0.0s

```
GCTGCTATCAGACACTCTTTTTTTAATCCACACAGAGACATATTGCCCGTTGCAGTCAGAATGAAAAGCT
```

Paradigmas de programación

```r
# Definimos la clase Secuencia en R
Secuencia <- function(cadena) { # Creamos una lista que actúa como clase
  obj <- list(cadena = cadena, n_A = 0, n_C = 0, n_G = 0, n_T = 0)

  calcular_conteos <- function() { # Método para calcular los conteos de A, C, G y T
    obj$n_A <<- sum(strsplit(obj$cadena, "")[[1]] == "A"); obj$n_C <<- sum(strsplit(obj$cadena, "")[[1]] == "C")
    obj$n_G <<- sum(strsplit(obj$cadena, "")[[1]] == "G"); obj$n_T <<- sum(strsplit(obj$cadena, "")[[1]] == "T")}

  describir <- function() { # Método para describir la secuencia y mostrar los conteos
    paste0("Secuencia: ", obj$cadena, "\n", "Conteo de A: ", obj$n_A, "\tConteo de C: ", obj$n_C, "\n",
                                            "Conteo de G: ", obj$n_G, "\tConteo de T: ", obj$n_T, "\n")}

  calcular_conteos() # Inicializamos calculando los conteos al crear el objeto

  obj$calcular_conteos <- calcular_conteos; obj$describir <- describir # Agregamos los métodos a la lista

  return(obj) # Devolvemos el objeto
}
```

Paradigmas de programación

Otros paradigmas

1. **Procedural**: estructura el comportamiento de un programa como una serie de **procedimientos** (funciones o subrutinas) que se llaman entre sí. El programa resultante se organiza como una jerarquía de llamadas entre estos procedimientos, donde cada uno realiza una tarea específica.

2. **Funcional:** los programas se describen como una **composición de funciones** más simples que transforman datos. El programa resultante **mapea valores de entrada a otros valores**.

Paradigmas de programación

```python
# Paradigma Procedural
entrada = list(range(10)); resultado = [] # Valores de entrada y la salida
for x in entrada:
    if x % 2 == 0:                        # Si el valor es par
        resultado.append(x ** 2)          # Aplicar cuadrado
print(resultado)
```

[23] ✓ 0.0s

```
[0, 4, 16, 36, 64]
```

```python
# Paradigma Funcional
resultado = list(map(lambda x: x ** 2,    # Generar cuadrado de los valores
                     filter(lambda x: x % 2 == 0,  # Filtrar valores por modulo 2 (pares)
                     range(10))))                  # Para esta entrada de valores
print(resultado)
```

[24] ✓ 0.0s

```
[0, 4, 16, 36, 64]
```

Paradigmas de programación

```python
tripletes = ["ATG", "GCT", "AAC", "TGC", "AGC", "GTT", "ACA", "TAA", "CAA", "AAG"]

# Enfoque procedural
complementos = {'A': 'T', 'T': 'A', 'C': 'G', 'G': 'C'}; resultado = []

for triplete in tripletes:    # Recorrer los tripletes
    if triplete[0] == 'A':    # Seleccionar tripletes que comienzan con A
        # Generar la secuencia rev_comp
        complementario = "".join([complementos[base] for base in triplete])
        resultado.append(complementario)
print(resultado)
```

[29] ✓ 0.0s

```
['TAC', 'TTG', 'TCG', 'TGT', 'TTC']
```

```python
resultado = list(map(lambda triplete: # Mapeado de con la función lambda
    "".join([complementos[base] for base in triplete]),  # Transformar a rev_comp
    filter(lambda triplete: triplete[0] == 'A', tripletes))) # Filtrado de tripletes
print(resultado)
```

[30] ✓ 0.0s

```
['TAC', 'TTG', 'TCG', 'TGT', 'TTC']
```

Paradigmas de programación

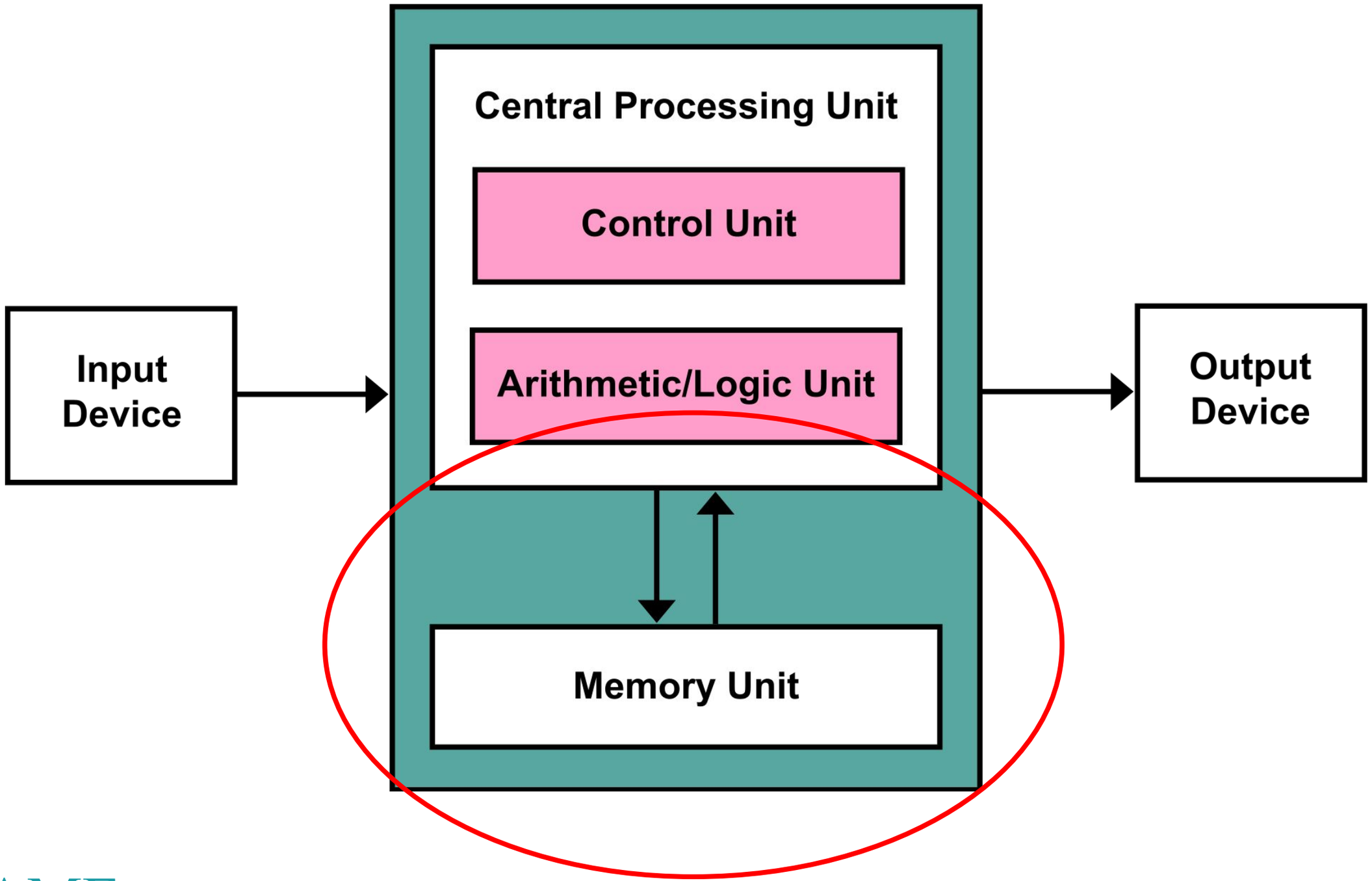
Aspecto	Procedural	Funcional
Enfoque	Resolución de problemas a través de pasos secuenciales y uso de funciones para dividir tareas.	Transformación de datos mediante funciones puras.
Estado	El estado del programa puede cambiar constantemente mediante variables.	Predica la inmutabilidad; los datos no cambian directamente.
Efectos secundarios	Las funciones pueden modificar variables globales o de otros contextos.	Las funciones no tienen efectos secundarios.
Reutilización de código	Usa funciones para evitar duplicación de código, pero con menor modularidad.	Promueve la reutilización y composibilidad de funciones.
Ejecución	Secuencial, depende del flujo de control (bucles, condicionales).	Basada en la composición y transformación de funciones.

Paradigmas de programación

Aspecto	Programación Imperativa	Programación Declarativa
Enfoque	Cómo se debe hacer (paso a paso).	Qué se quiere lograr (resultado deseado).
Control de Flujo	El programador define explícitamente el orden de ejecución.	El sistema decide el orden, basado en el resultado.
Estado	Usa variables y estado mutable.	Favorece la inmutabilidad y evita efectos secundarios.
Ejecución	Secuencial y dependiente de instrucciones.	Evaluación basada en expresiones.

https://en.wikipedia.org/wiki/Comparison_of_multi-paradigm_programming_languages

Tipos de variables



Tipos de variables: python

Tipo de Variable	Descripción
`int`	Números enteros, positivos o negativos, sin decimales.
`float`	Números de punto flotante (decimales).
`str`	Cadenas de texto.
`bool`	Valores lógicos, `True` o `False`.
`list`	Lista ordenada de elementos, que pueden ser de diferentes tipos.
`tuple`	Tupla ordenada e inmutable de elementos.
`dict`	Diccionario que almacena pares clave-valor.
`set`	Conjunto no ordenado de elementos únicos.

```
# Inicialización (con asignación); Asignación; Cambio de valor
integer = 25; integer = 0; integer += 5; print(integer)
```

✓ 0.0s

```
5
```

Tipos de variables: python

```python
from Bio import Entrez, SeqIO

# Acceder a genbank, sequencia NC_045512.2
raw_record = Entrez.efetch(db="nucleotide", id="NC_045512.2", rettype="gb", retmode="text")
registro = SeqIO.read(raw_record, "genbank"); print(type(registro))

# String - Descripción de la secuencia
print(f"{type(registro.description)}\t{registro.description}")

# Lista - Listar los identificadores de las características
lista_features = [str(feature.type) for feature in registro.features]
print(f"{type(lista_features)}\t{lista_features}")

# Integer - Longitud total de la secuencia
print(f"{type(len(registro.seq))}\t{len(registro.seq)}")
```

✓ 1.1s

```
<class 'Bio.SeqRecord.SeqRecord'>
<class 'str'>   Severe acute respiratory syndrome coronavirus 2 isolate Wuhan-Hu-1, complete genome
<class 'list'>  ['source', "5'UTR", 'gene', 'CDS', 'mat_peptide', 'mat_peptide', 'mat_peptide', 'mat
<class 'int'>   29903
```

https://www.ncbi.nlm.nih.gov/nuccore/1798174254

Tipos de variables: python

```python
# Diccionario - Diccionario con genes y sus posiciones
genes_y_posiciones = {}
for feature in registro.features:
    if feature.type == "gene":
        genes_y_posiciones[feature.qualifiers.get(
            "gene", ["desconocido"])[0]] = (
                str(feature.location),
                feature.qualifiers.get("gene_synonym", ["Desconocido"])[0],
                feature.qualifiers.get("locus_tag", ["Desconocido"])[0])

print(type(genes_y_posiciones)); print(genes_y_posiciones)
```

✓ 0.0s

```
<class 'dict'>
{'ORF1ab': ('[265:21555](+)', 'Desconocido', 'GU280_gp01'), 'S': ('[21562:25384](+)',
```

https://www.ncbi.nlm.nih.gov/nuccore/1798174254

Tipos de variables: python

```python
# Booleano - Verificar si hay un gen de interés en el archivo
print(type("S" in genes_y_posiciones)); print("S" in genes_y_posiciones)

# Tupla - Acceder a la información del gen de interés
print(type(genes_y_posiciones['S'])); print(genes_y_posiciones['S'])

# Conjunto (set) - Conjunto de los nucleótidos únicos en la secuencia
print(type(set(registro.seq))); print(set(registro.seq))
```

✓ 0.0s

```
<class 'bool'>
True
<class 'tuple'>
('[21562:25384](+)', 'spike glycoprotein', 'GU280_gp02')
<class 'set'>
{'T', 'G', 'C', 'A'}
```

https://www.ncbi.nlm.nih.gov/nuccore/1798174254

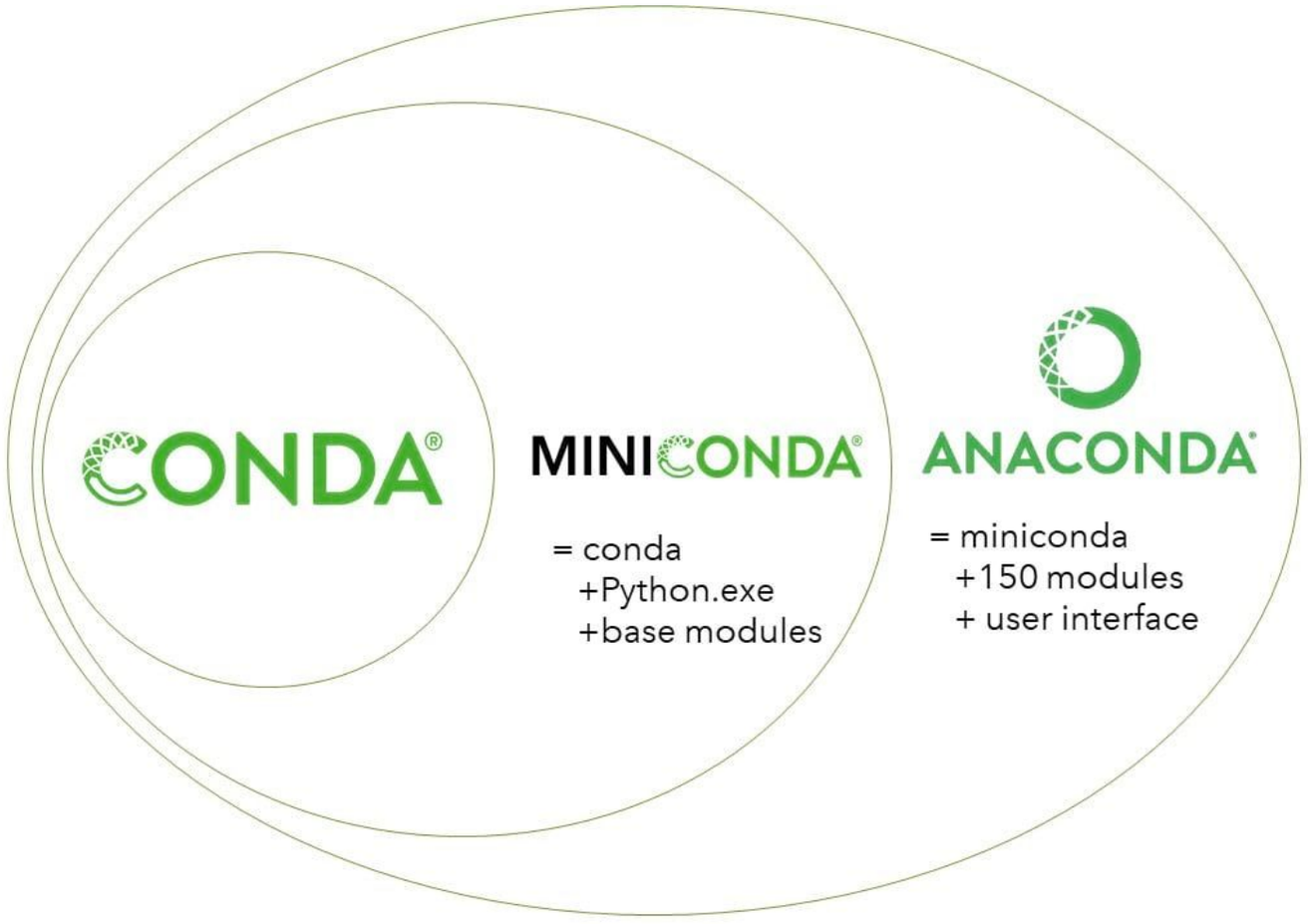
Tipos de variables: python

Tipo de Variable	Descripción	Características Principales	Paquete
`DataFrame`	Estructura de datos tabular bidimensional con etiquetas en filas y columnas.	- Permite almacenar diferentes tipos de datos (numeros, texto, booleanos). - Tiene etiquetas para filas y columnas. - Funcionalidad avanzada como selección, filtrado, y operaciones por etiquetas.	`pandas`
`Array`	Estructura de datos multidimensional que almacena elementos del mismo tipo.	- Todos los elementos deben ser del mismo tipo. - Más eficiente en operaciones matemáticas y procesamiento de datos numéricos. - No tiene etiquetas para filas o columnas.	`numpy`

Tipos de variables: R

Tipo de Variable	Descripción
`numeric`	Números reales, incluyendo decimales.
`integer`	Números enteros. Se definen con la función `as.integer()` o el sufijo `L`.
`character`	Cadenas de texto.
`logical`	Valores lógicos: `TRUE` o `FALSE`.
`factor`	Datos categóricos que toman un conjunto limitado de valores distintos.
`list`	Colección heterogénea de elementos de diferentes tipos.
`vector`	Colección homogénea de elementos (todos del mismo tipo).
`matrix`	Arreglo bidimensional de elementos homogéneos.
`array`	Arreglo multidimensional de elementos homogéneos.
`data.frame`	Tabla de datos bidimensional donde las columnas pueden tener diferentes tipos.

Instalación de Software: conda



Miniconda: https://docs.anaconda.com/miniconda/

Tutorial:

Windows: https://docs.conda.io/projects/conda/en/latest/user-guide/install/windows.html

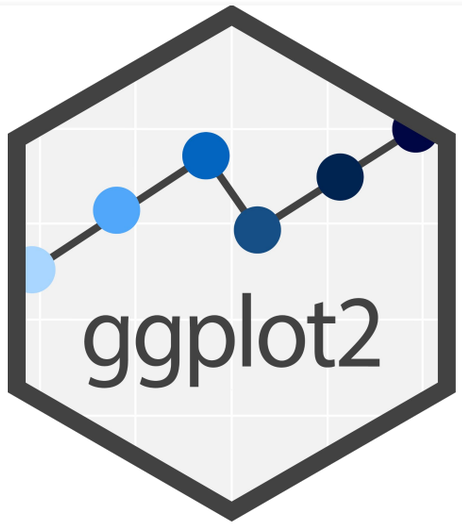
macOS: https://docs.conda.io/projects/conda/en/latest/user-guide/install/macos.html

Linux: https://docs.conda.io/projects/conda/en/latest/user-guide/install/linux.html

Ejemplos prácticos

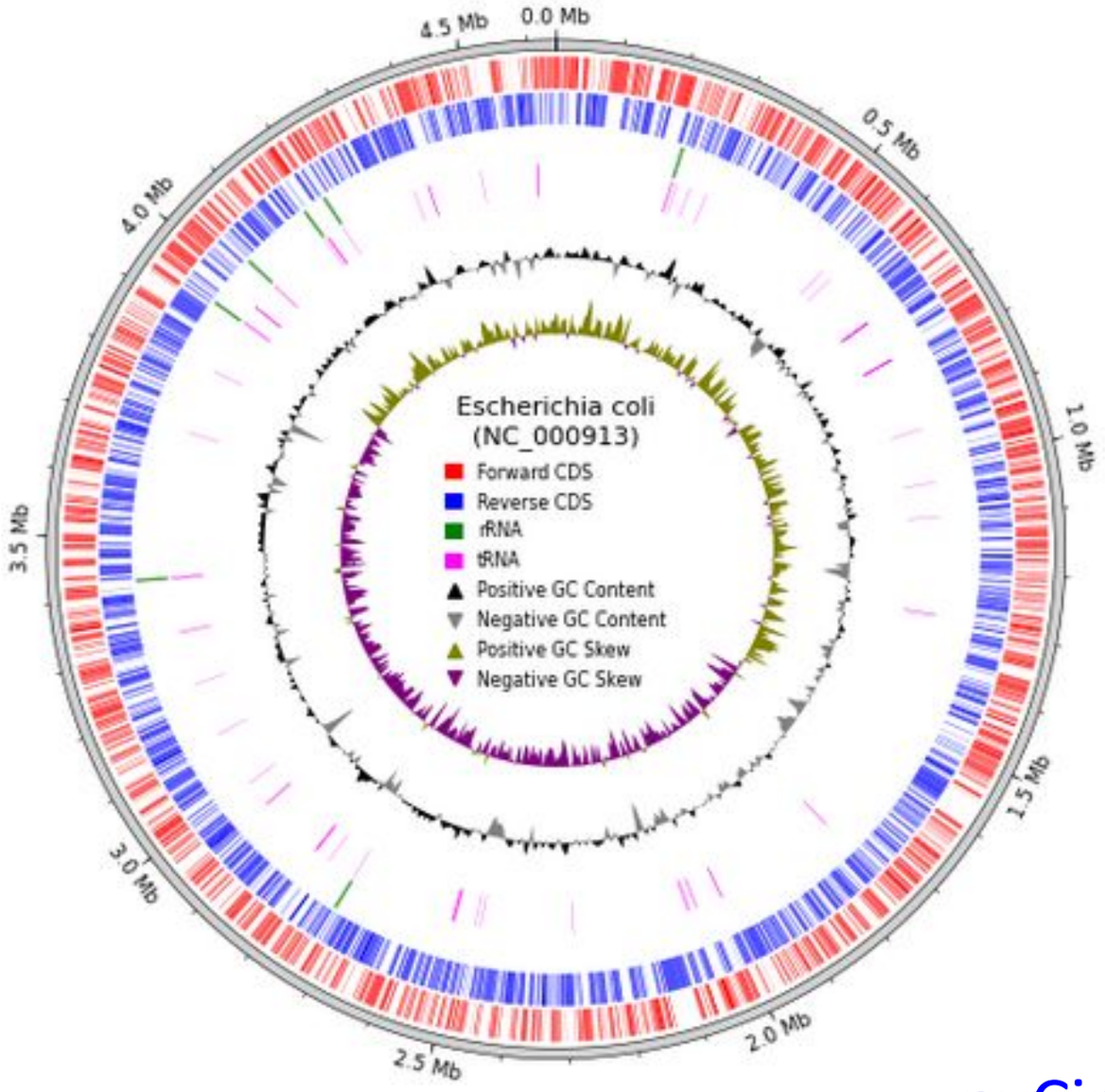
1)

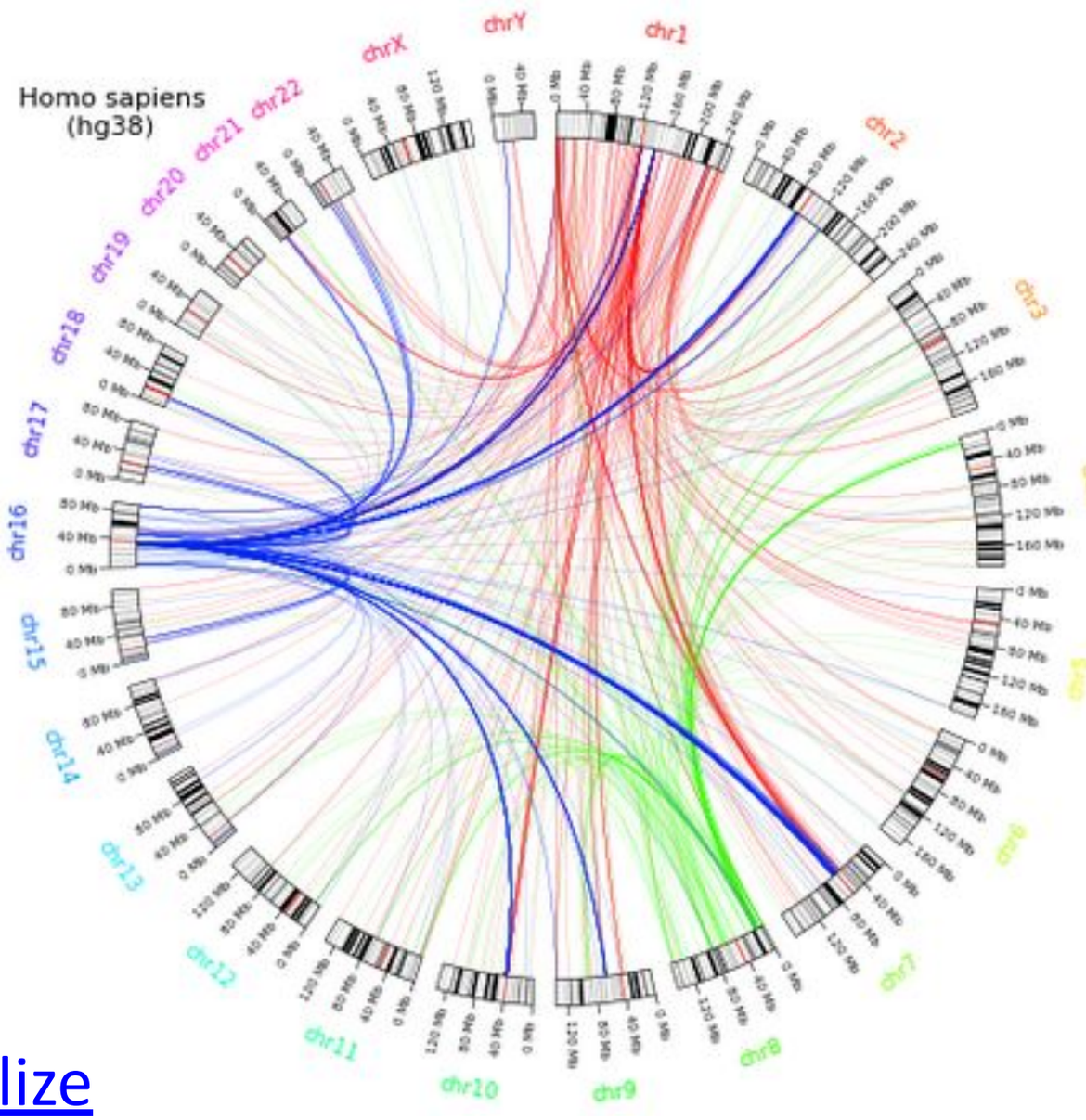
2)



https://github.com/mirodriguezgal/MBIOINDI

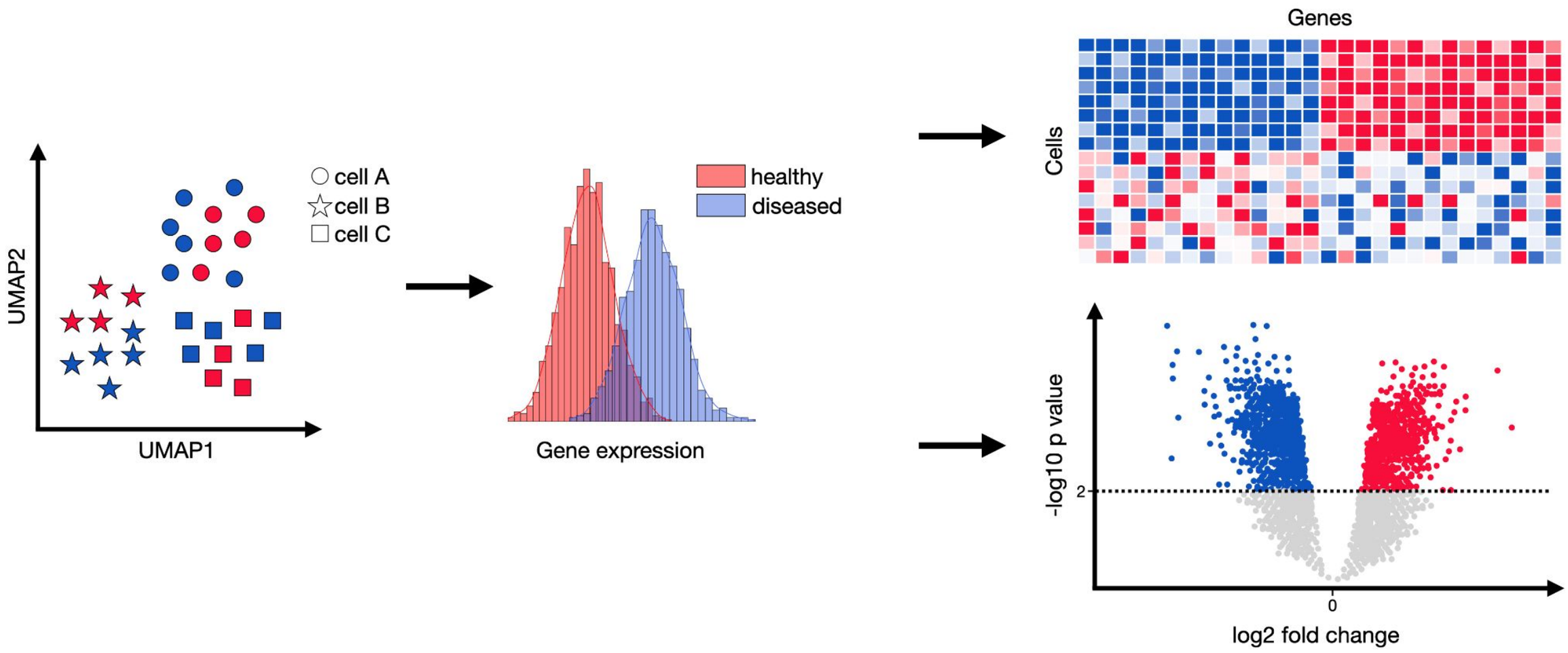
Introducción a la Visualización de Datos



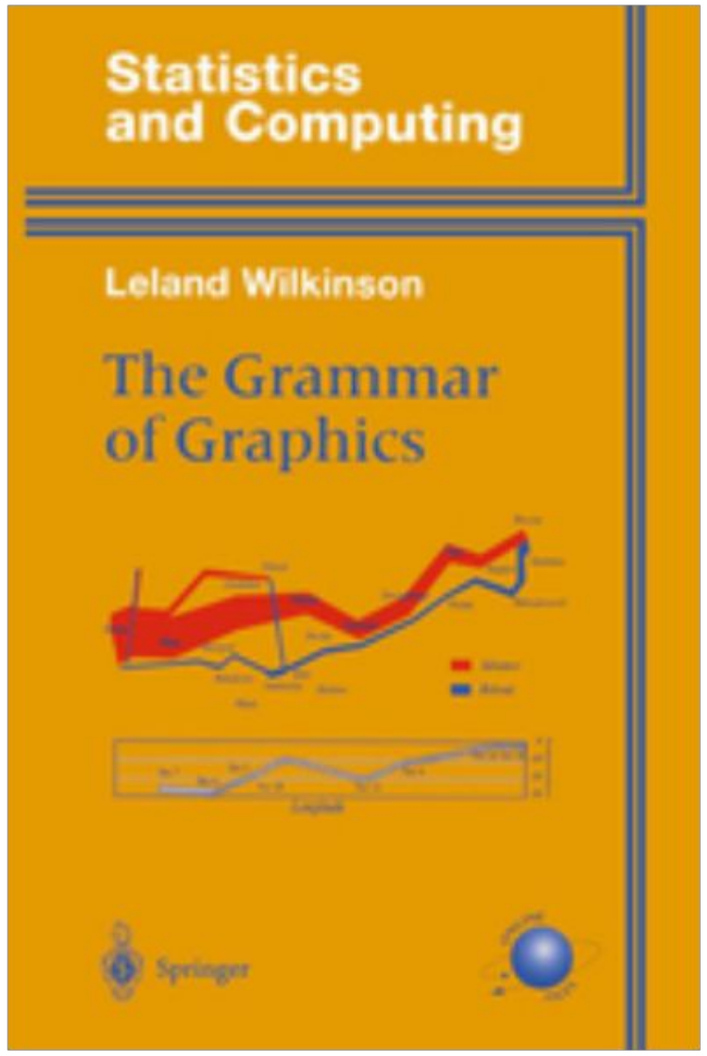


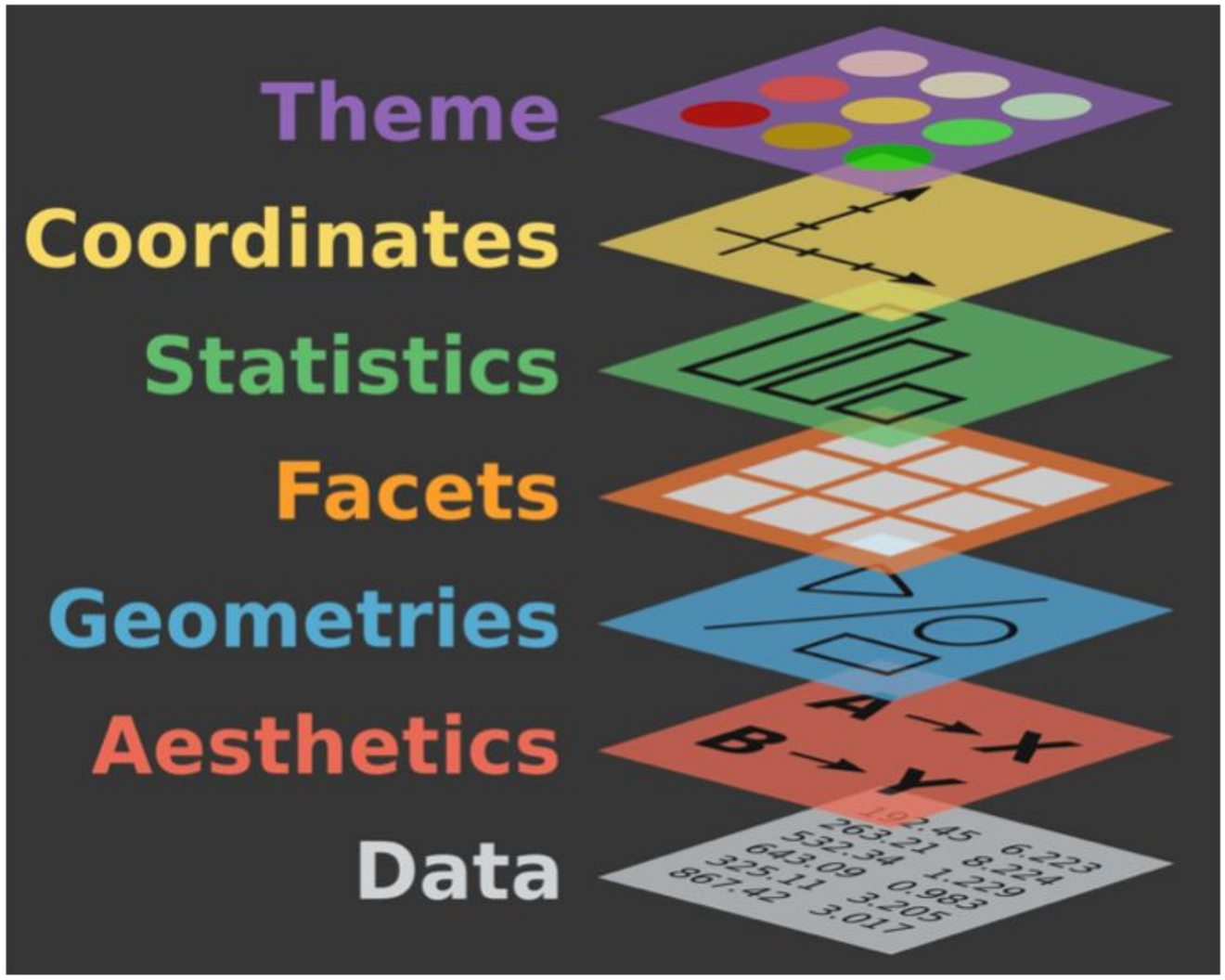
[pyCirclize]

Introducción a la Visualización de Datos



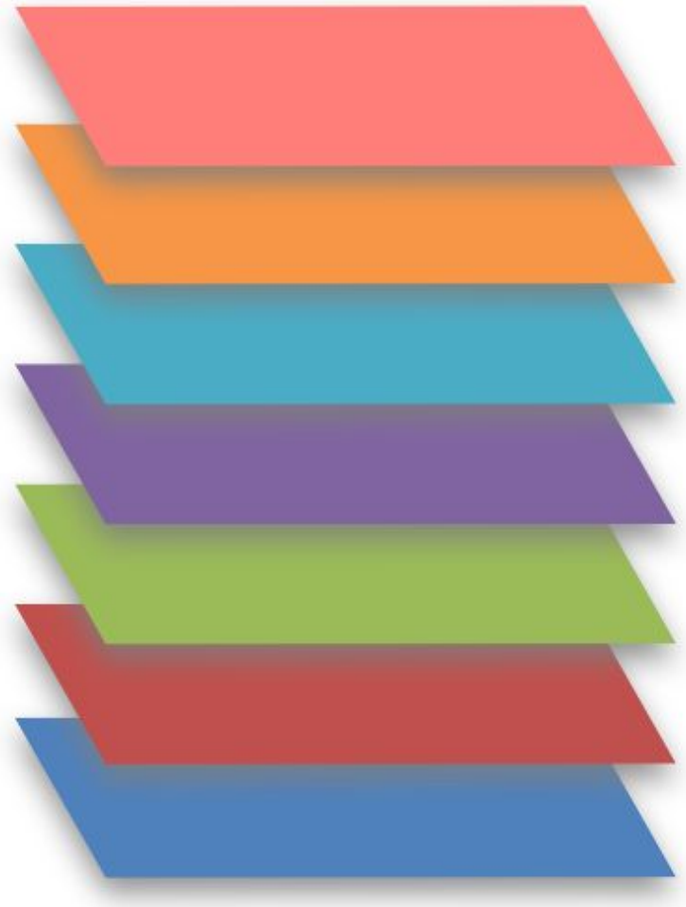
La gramática de los gráficos



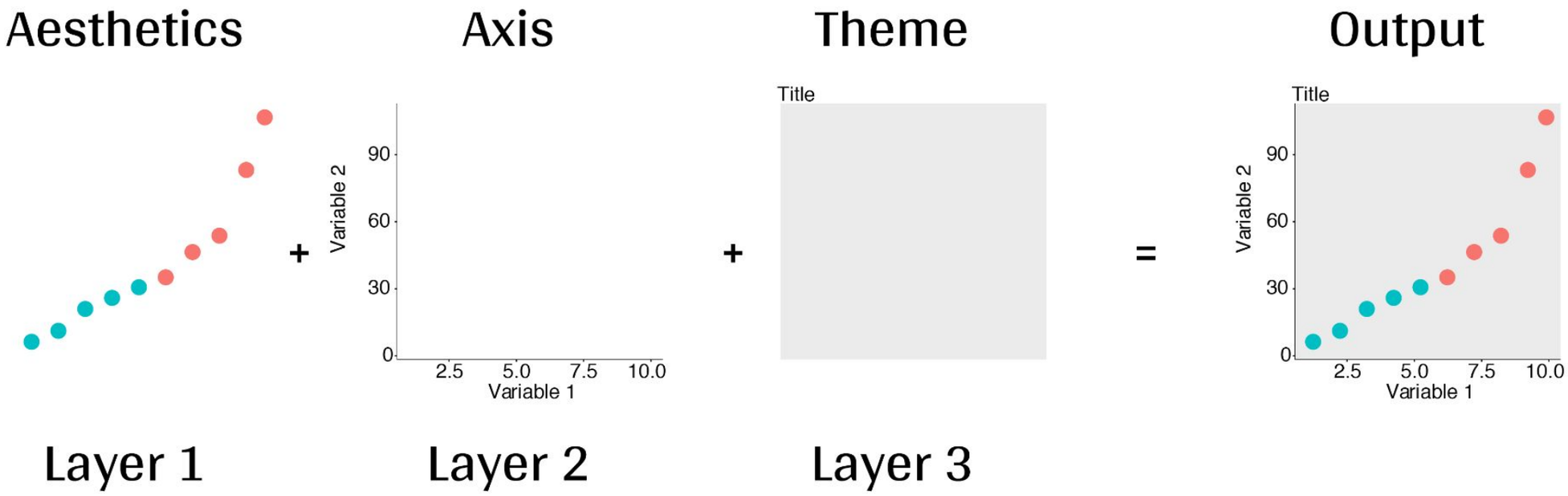


La gramática de los gráficos

Describes all the non-data ink	Theme
Plotting space for the data	Coordinates
Statistical models & summaries	Statistics
Rows and columns of sub-plots	Facets
Shapes used to represent the data	Geometries
Scales onto which data is mapped	Aesthetics
The actual variables to be plotted	Data



La gramática de los gráficos



Formatos de tablas de datos

site	species_a	species_b	species_c
site1	11	54	0
site2	23	17	10
site3	7	0	33

CORTO

site	species	count
site1	species_a	11
site1	species_b	54
site1	species_c	0
site2	species_a	23
site2	species_b	17
site2	species_c	10
site3	species_a	7
site3	species_b	0
site3	species_c	33

LARGO

ggplot2

Basics

ggplot2 is based on the **grammar of graphics**, the idea that you can build every graph from the same components: a **data** set, a **coordinate system**, and **geoms**—visual marks that represent data points.

To display values, map variables in the data to visual properties of the geom (**aesthetics**) like **size**, **color**, and **x** and **y** locations.

Complete the template below to build a graph.

```
ggplot (data = <DATA>) +                                   ] required
<GEOM_FUNCTION>(mapping = aes(<MAPPINGS>),                 ]
 stat =<STAT>, position =<POSITION>) +                     ] Not required, sensible defaults supplied
<COORDINATE_FUNCTION> +                                    ]
<FACET_FUNCTION> +                                         ]
<SCALE_FUNCTION> +                                         ]
<THEME_FUNCTION>                                           ]
```

ggplot(data = mpg, **aes(**x = cty, y = hwy**))** Begins a plot that you finish by adding layers to. Add one geom function per layer.

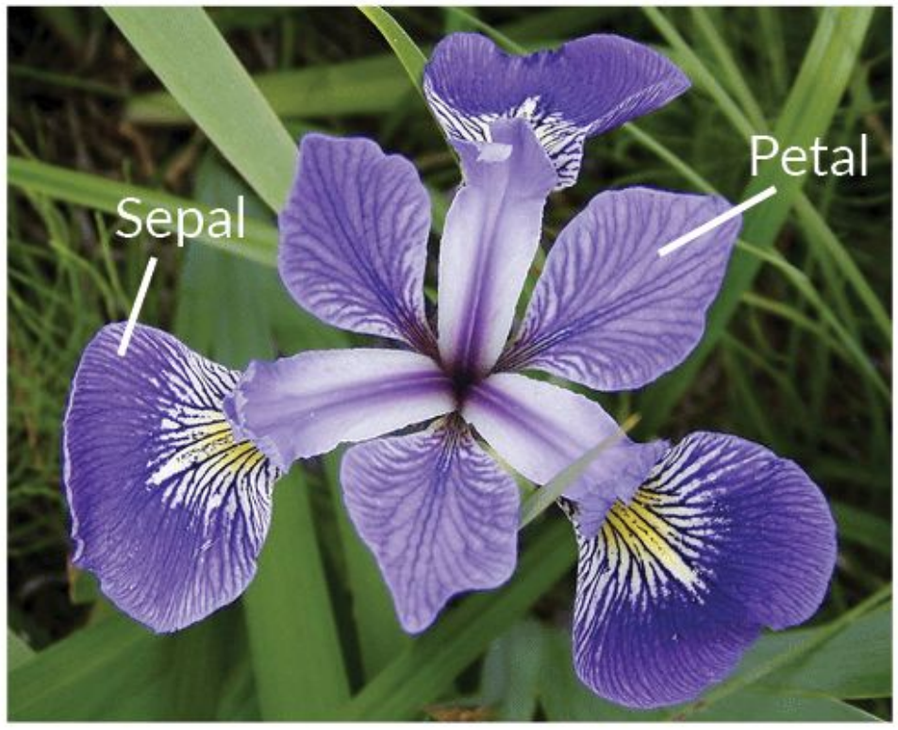
last_plot() Returns the last plot.

ggsave("plot.png", width = 5, height = 5**)** Saves last plot as 5' x 5' file named "plot.png" in working directory. Matches file type to file extension.

Geometrías comunes

Geometría	Uso
`geom_point()`	Gráficos de dispersión (scatterplots).
`geom_line()`	Líneas conectando puntos.
`geom_bar()`	Diagramas de barras.
`geom_histogram()`	Histogramas.
`geom_boxplot()`	Diagramas de caja (boxplots).
`geom_density()`	Curvas de densidad.
`geom_violin()`	Diagramas de violín (variación de boxplot que muestra la densidad).
`geom_tile()`	Heatmaps (gráficos de calor).

Dataset Iris



Iris Versicolor

Iris Setosa

Iris Virginica

Gráfico de dispersión (scatterplot)

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width)) + geom_point()
```

Datos | Estéticas básicas (x, y) | Geometría

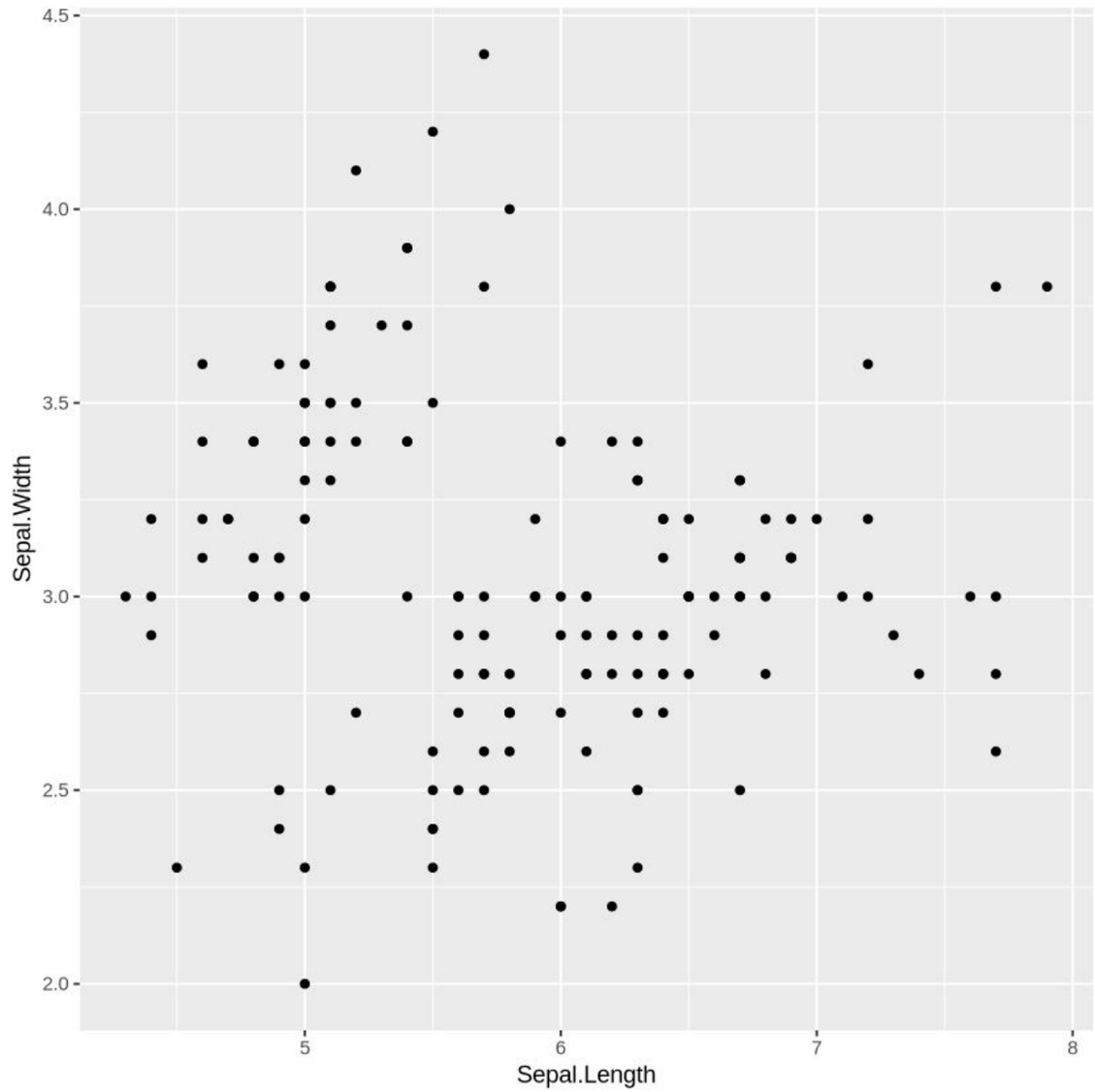


Gráfico de dispersión (scatterplot)

```
ggplot(iris, aes(x = Sepal.Length, y = Sepal.Width, color = Species)) + geom_point()
```

Datos | Estéticas (x, y, color) | Geometría

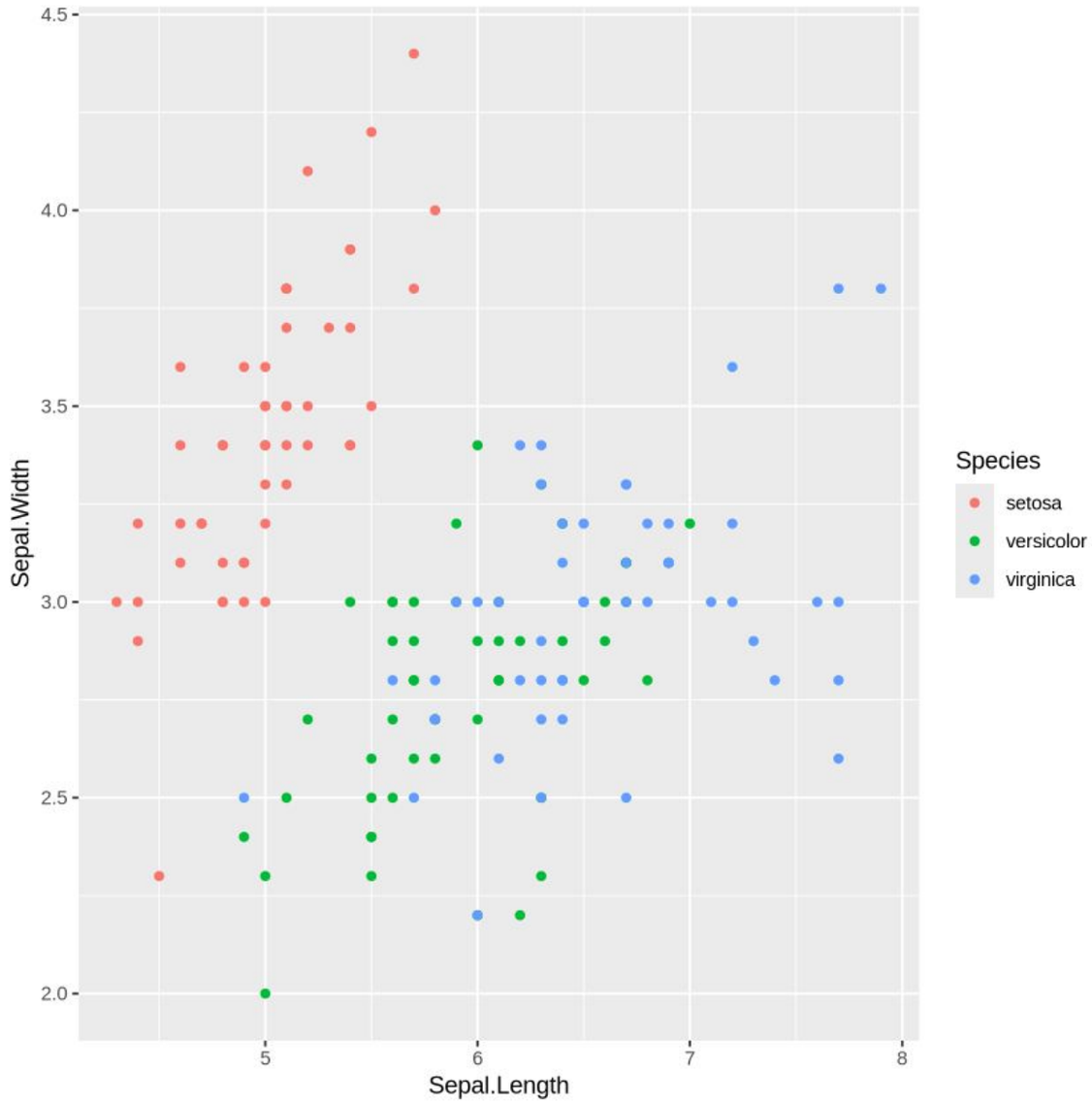


Gráfico de dispersión (scatterplot)

```r
ggplot(iris,
aes(x = Sepal.Length, y = Petal.Length,
  color = Species)) +
  geom_point(alpha = 0.7, size = 3) +
  geom_smooth(method = "lm", se = TRUE,
  linetype = "dashed") +
  facet_wrap(~Species) +
  labs(title = "Sépalo vs pétalo",
    x = "Largo del sépalo (cm)",
    y = "Largo del pétalo (cm)") +
  theme_minimal()
```

Datos

Estéticas

Geometría

Estadísticas

Facetas

Temas

Gráfico de dispersión (scatterplot)

```r
ggplot(iris,
aes(x = Sepal.Length, y = Petal.Length,
  color = Species)) +
  geom_point(alpha = 0.7, size = 3) +
  geom_smooth(method = "lm", se = TRUE,
  linetype = "dashed") +
  facet_wrap(~Species) +
  labs(title = "Sépalo vs pétalo",
    x = "Largo del sépalo (cm)",
    y = "Largo del pétalo (cm)") +
  theme_minimal()
```

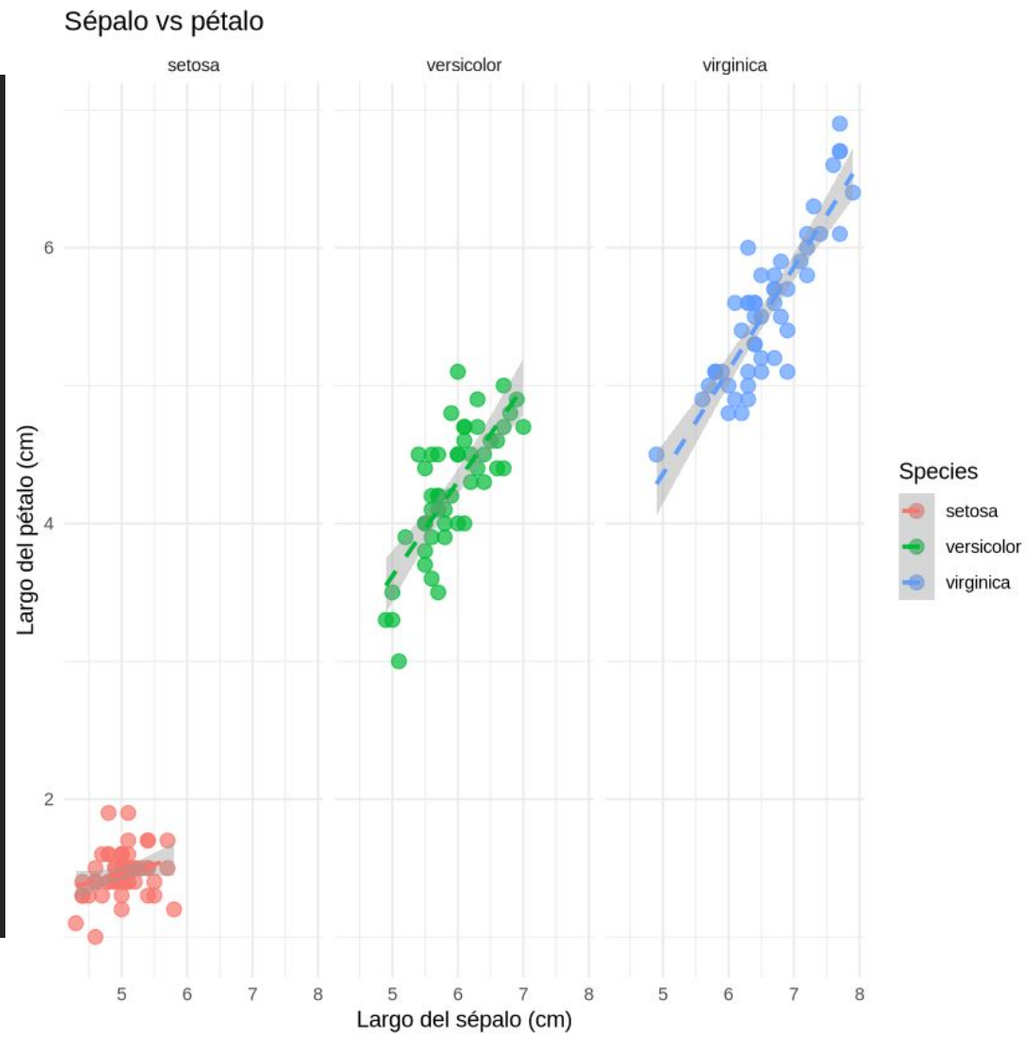


Gráfico de dispersión (scatterplot)

```r
ggplot(DNase,
aes(x = conc, y = density,
color = Run)) +
geom_point() +
geom_smooth(method = "loess",
 se = TRUE) +
scale_x_log10() +
facet_wrap(~Run) +
labs(y = "Densidad óptica",
 x = "[S] (log10)",
 title = "[S] vs Densidad óptica") +
theme_minimal()
```

Datos

Estéticas

Geometría

Estadísticas

Facetas

Coordenadas

Temas

Gráfico de dispersión (scatterplot)

```r
ggplot(DNase,
aes(x = conc, y = density,
color = Run)) +
geom_point() +
geom_smooth(method = "loess",
 se = TRUE) +
scale_x_log10() +
facet_wrap(~Run) +
labs(y = "Densidad óptica",
 x = "[S] (log10)",
 title = "[S] vs Densidad óptica") +
theme_minimal()
```

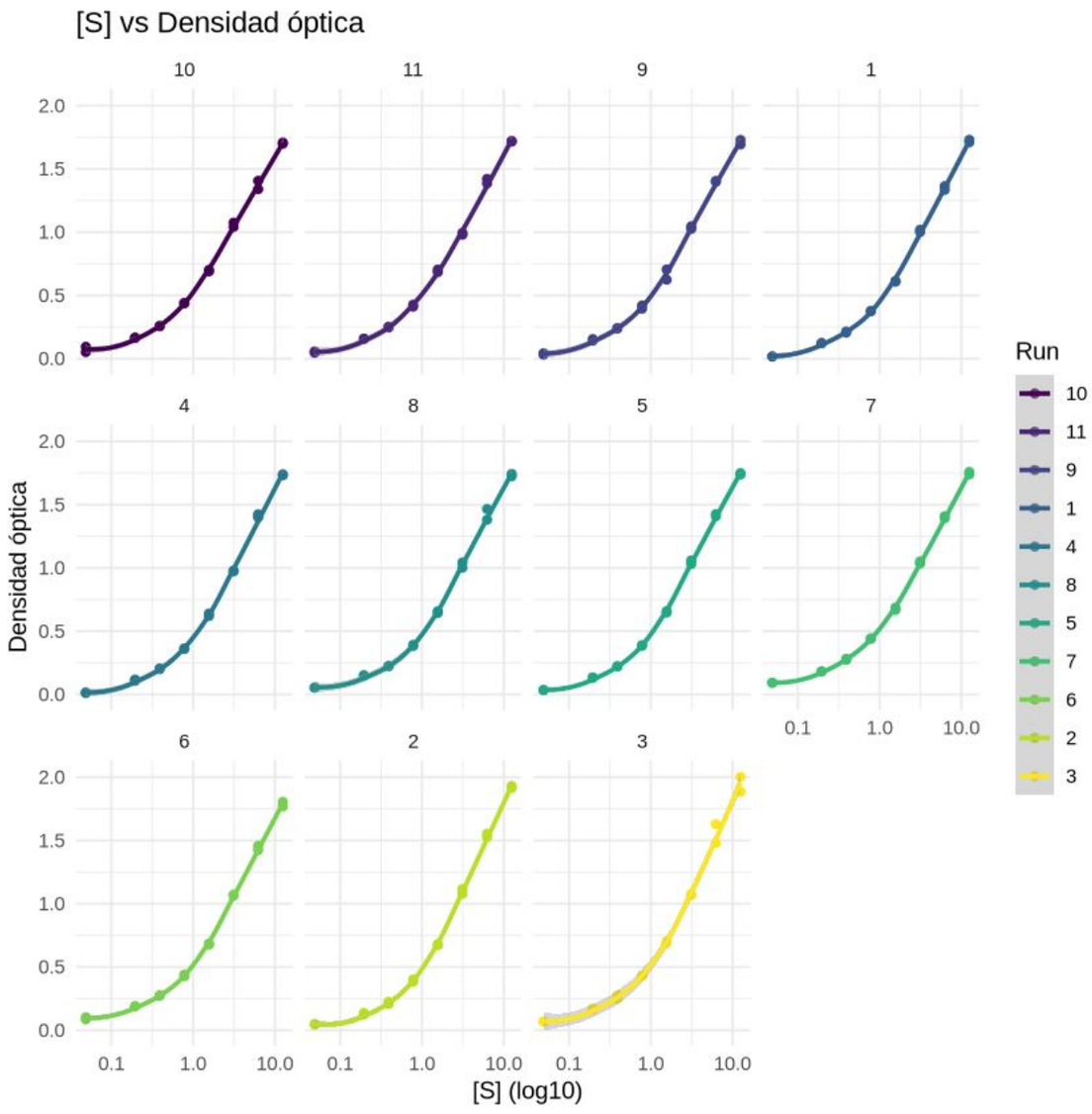
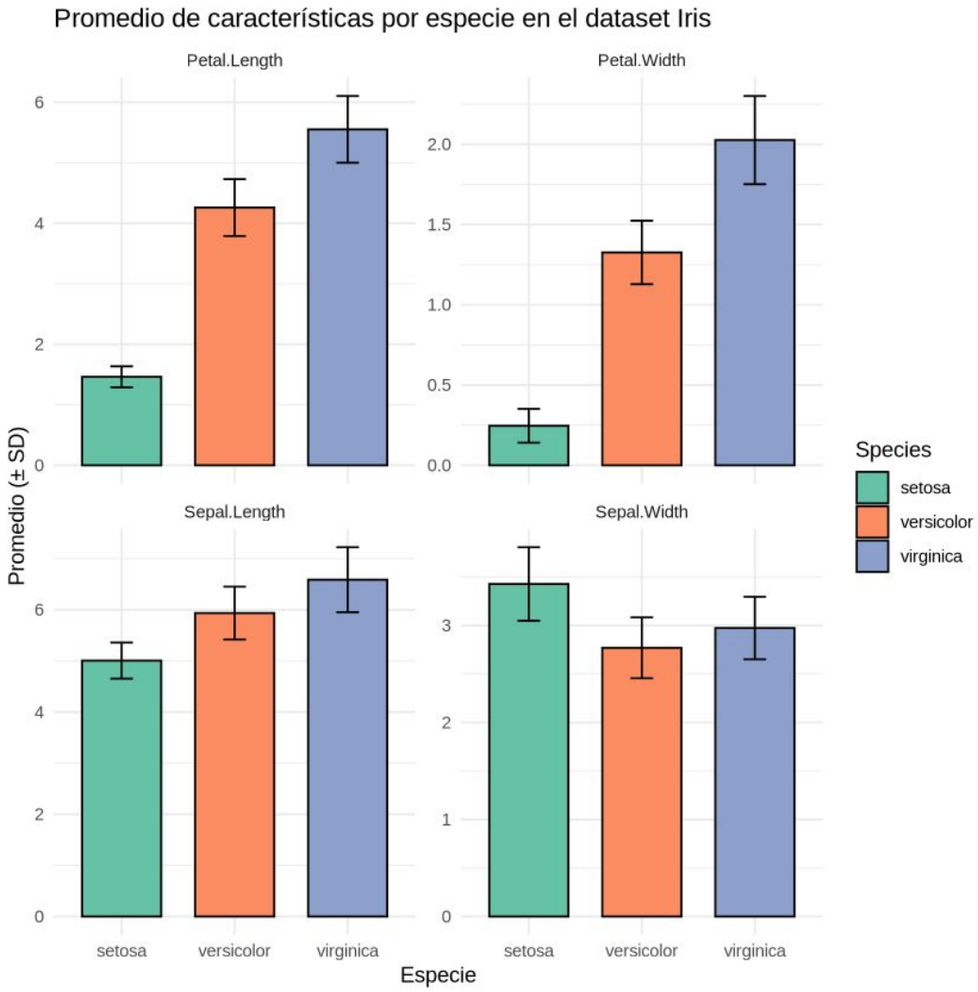


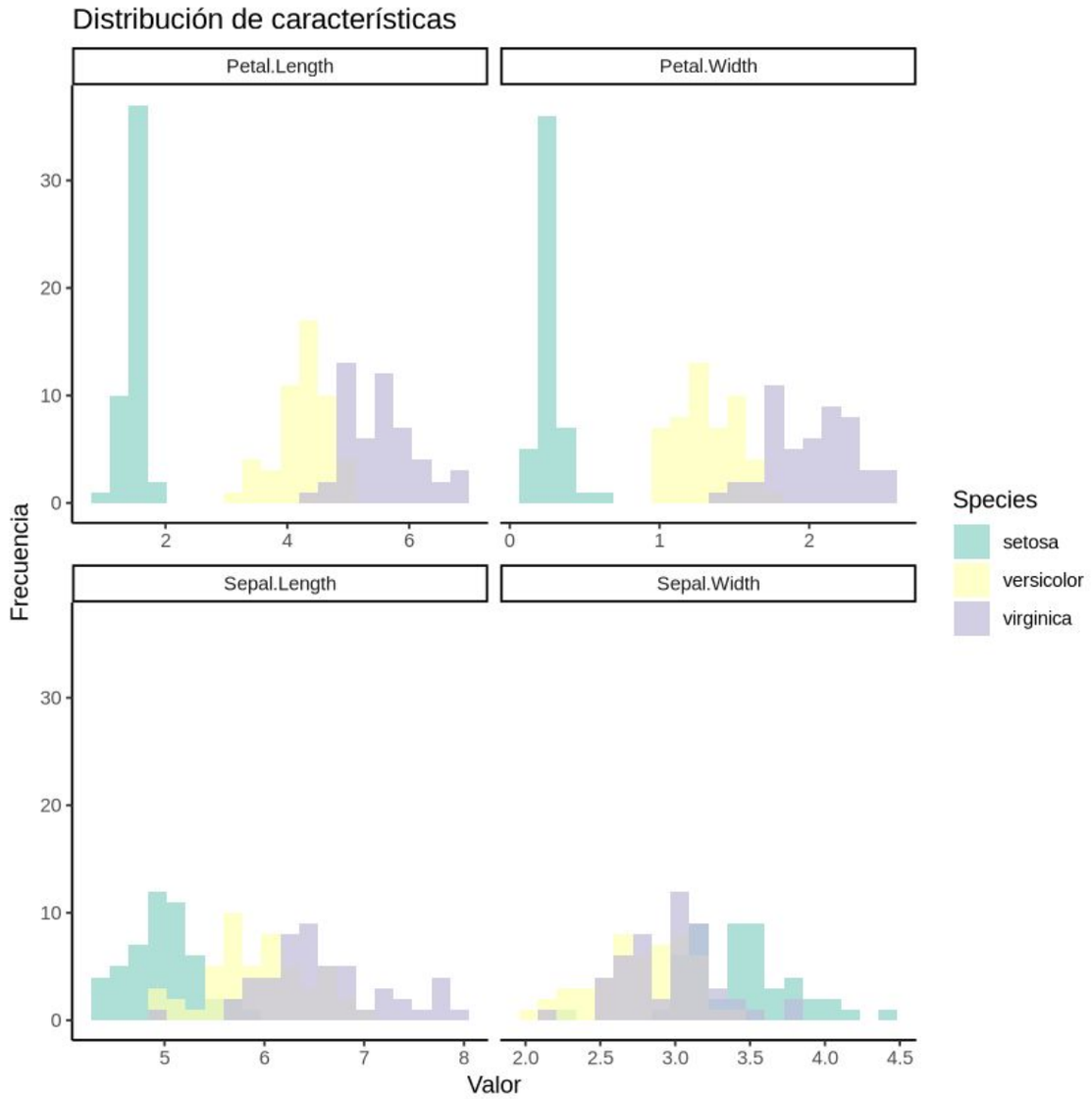
Gráfico de barras (Bar Plot)

```
ggplot(iris_summary,
aes(x = Species, y = mean_value, fill = Species)) +
geom_bar(stat = "identity", position = "dodge",
 color = "black", width = 0.7) +
geom_errorbar(aes(ymin = mean_value - sd_value,
 ymax = mean_value + sd_value),
 position = position_dodge(width = 0.7),
 width = 0.2,
 color = "black") +
facet_wrap(~ Feature, scales = "free_y") +
labs(title = "Promedios en Iris",
  x = "Especie",
    y = "Promedio (± SD)"
  ) + theme_minimal() +
  scale_fill_brewer(palette = "Set2")
```



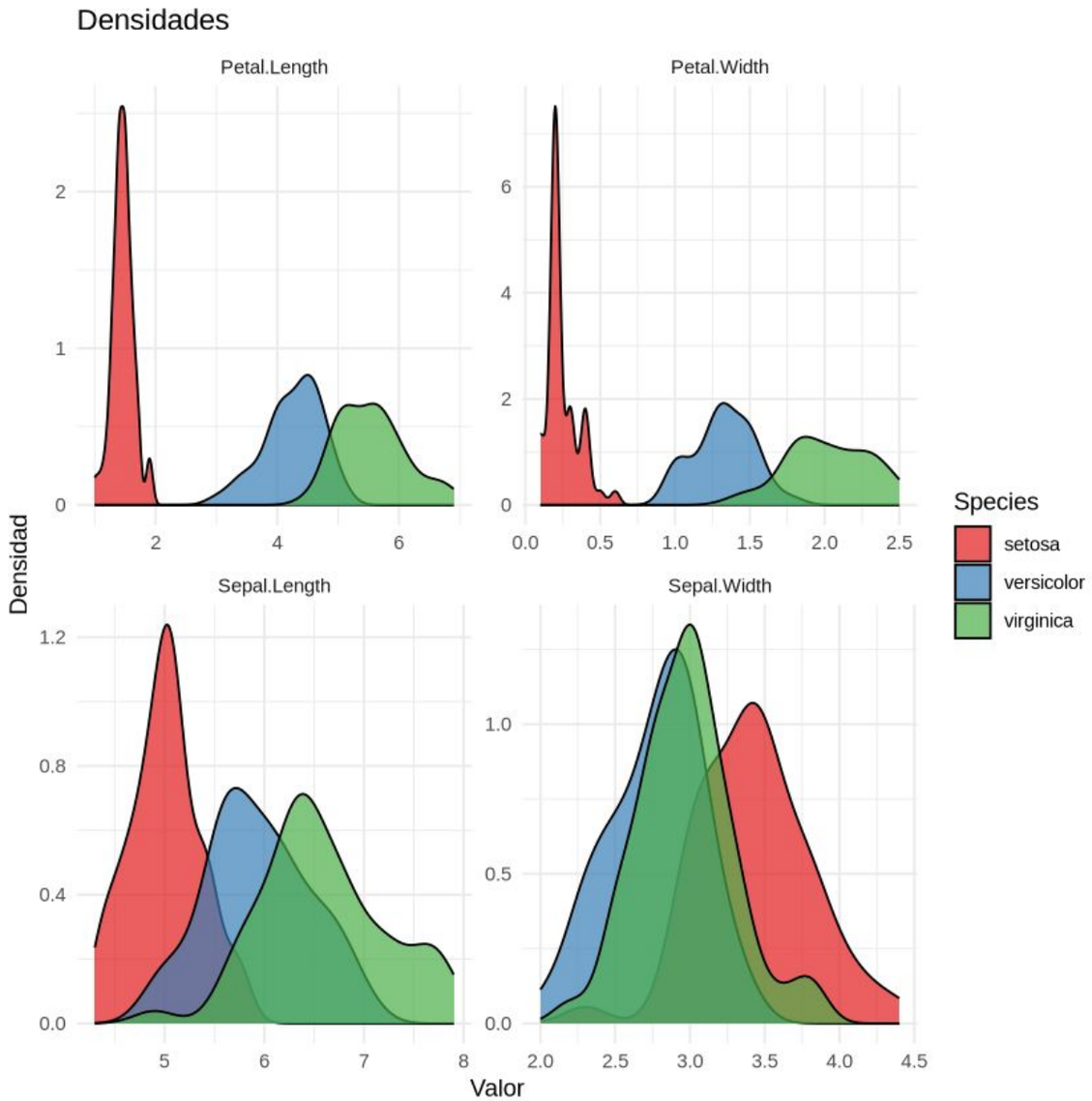
Distribuciones: histogramas

```
ggplot(iris_long,
aes(x = Value, fill = Species)) +
geom_histogram(
  position = "identity",
  alpha = 0.7, bins = 20) +
facet_wrap(~ Feature,
 scales = "free_x") +
labs(title = "Histogramas de Iris",
  x = "Valor", y = "Frecuencia") +
theme_classic() +
scale_fill_brewer(palette = "Set3")
```



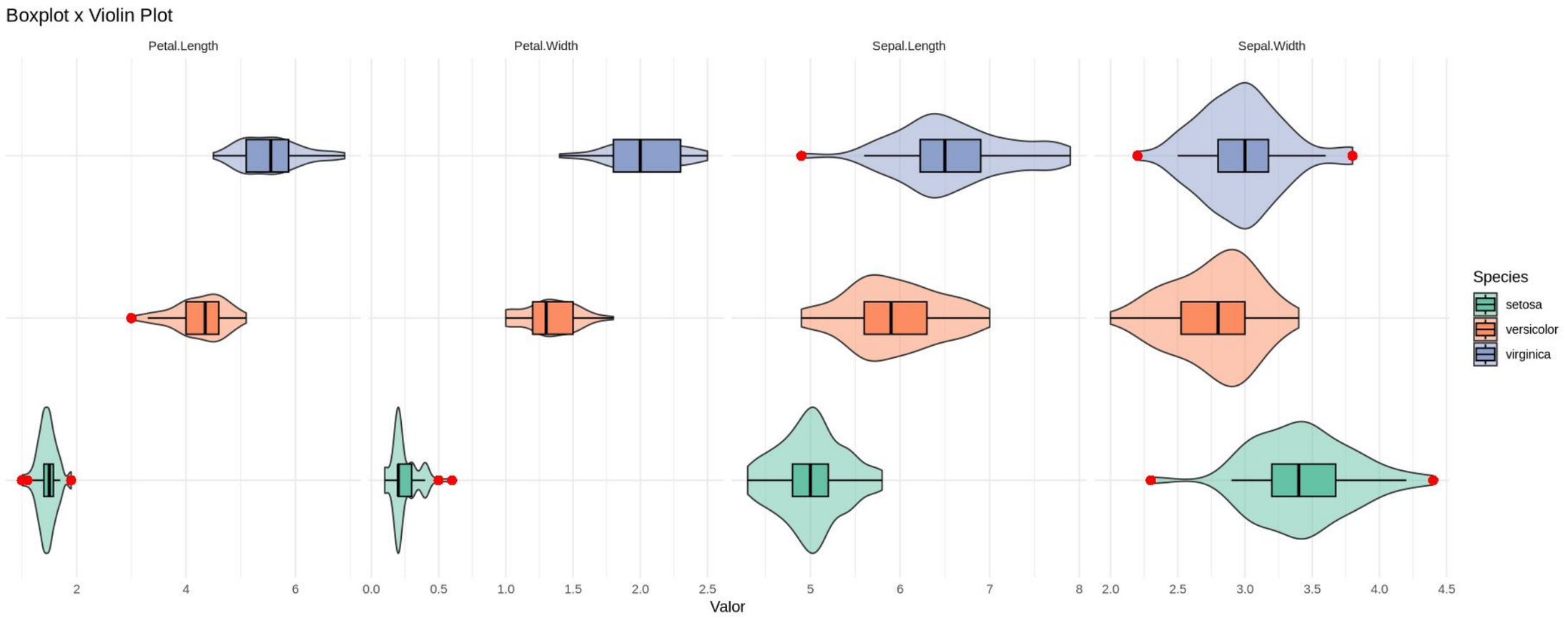
Distribuciones: densidad

```
ggplot(iris_long, aes(x = Value, fill = Species)) + geom_density(alpha = 0.7) + facet_wrap(~ Feature, scales = "free") +
labs(title = "Densidades", x = "Valor", y = "Densidad") + theme_minimal() + scale_fill_brewer(palette = "Set1")
```



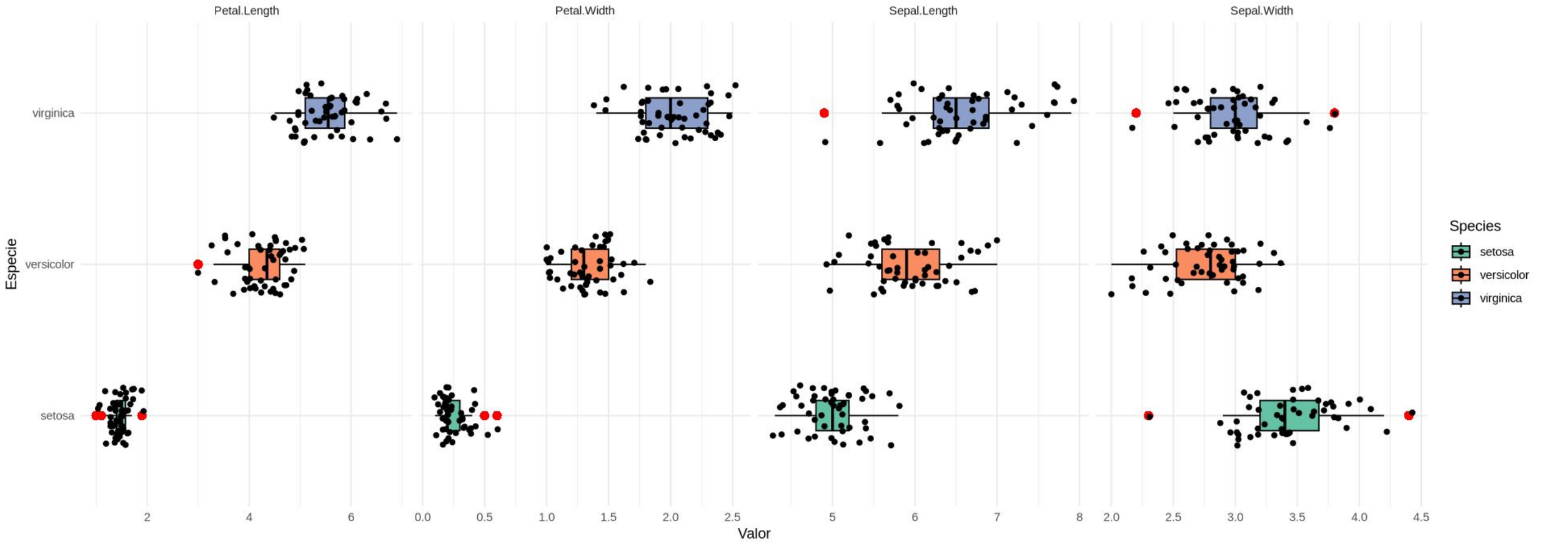
Distribuciones: cajas y violines

```r
ggplot(iris_long, aes(x = Species, y = Value, fill = Species)) +
  geom_violin(alpha = 0.5) +  # Actua similar al de densidad
  geom_boxplot(width = 0.2, color = "black", outlier.shape = NA) +
  facet_wrap(~ Feature, scales = "free") +
  labs( title = "Boxplot x Violin Plot", x = "Especie", y = "Valor") +
  theme_minimal() + scale_fill_brewer(palette = "Set2")
```



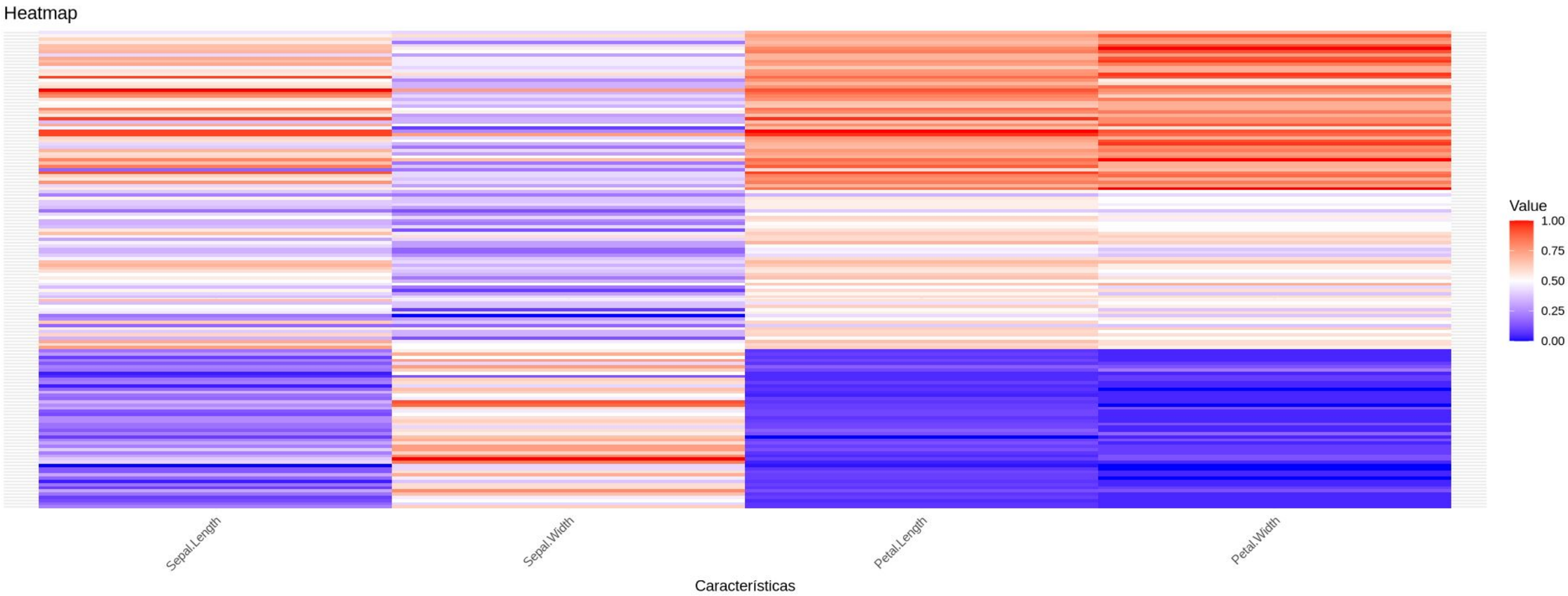
Distribuciones: cajas y ‘jitter’

```r
ggplot(iris_long, aes(x = Species, y = Value, fill = Species)) +
  geom_boxplot(width = 0.2, color = "black",
    outlier.shape = 16, outlier.colour = "red", outlier.size = 3) +
  geom_jitter(position = position_jitter(0.2)) +
  facet_wrap(~ Feature, scales = "free_x", ncol = 4) + coord_flip() +
  labs( title = "Boxplot x Jitter Plot", x = "Especie", y = "Valor") +
  theme_minimal() + scale_fill_brewer(palette = "Set2")
```



Mapas de calor (heatmap)

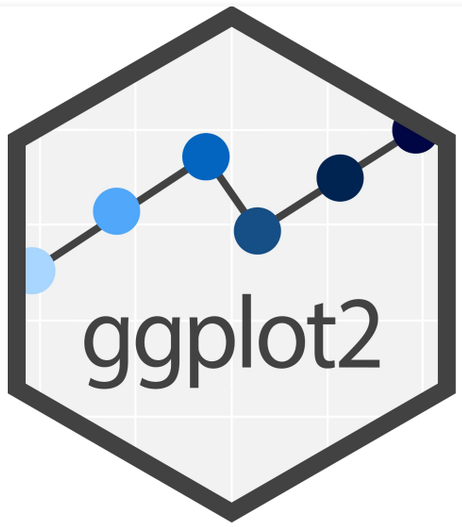
```
ggplot(ma_long, aes(x = Variable, y = Species, fill = Value)) + geom_tile() + scale_fill_gradient2(low = "blue",
  high = "red", mid = "white", midpoint = 0.5) + theme_minimal() + theme(axis.text.x = element_text(angle = 45,
  hjust = 1), axis.text.y = element_blank()) + labs(title = "Heatmap", x = "Características", y = "")
```



Ejemplos prácticos

1)

2)



https://github.com/mirodriguezgal/MBIOINDI