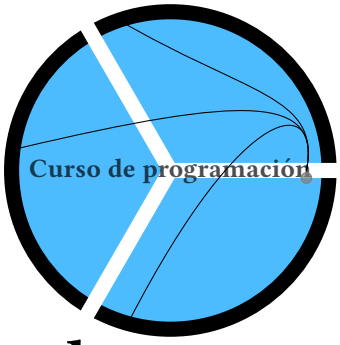

Python
R con Rstudio
Automatización *N8N*

FACULTAD DE CIENCIAS Y BIOTECNOLOGÍA

2025

Revisión 3

Juan Pablo Lopera Vélez
Juan Gustavo Diosa Peña
John J. Estrada Álvarez

01/17/2025

Aplicación Virtual para programar en R con Rstudio

El siguiente enlace permite acceder al recurso para iniciar las practicas de aprendizaje UCES:	https://jestrada2020.github.io/ProgramacionRRstudiov1/

También puedes usar el código QR para acceder al recurso para iniciar las practicas de aprendizaje UCES:

Enlaces claves del curso

Aplicación Virtual para las clases de programación día a día en el aula Bloq-B 505

El siguiente enlace permite acceder al recurso para iniciar las practicas de aprendizaje UCES:	https://jestrada2020.github.io/principiosdeprogramacionpythonV1/

También puedes usar el código QR para acceder al recurso para iniciar las practicas de aprendizaje UCES:

Aplicación Virtual para programar en Shiny for Python - Google Colab

El siguiente enlace permite acceder al recurso para iniciar las practicas de aprendizaje UCES:	https://jestrada2020.github.io/ProgramacionShinyForPythonV1/

También puedes usar el código QR para acceder al recurso para iniciar las practicas de aprendizaje UCES:

CLAVE MATRICULACIÓN AULA VIRTUAL

BIOPYTHONS1

Índice general

Introducción

Uno de los principales beneficios de conocer un lenguaje de programación es la capacidad de analizar datos de manera eficiente. Herramientas como Python y R permiten a los científicos procesar grandes conjuntos de datos, realizar análisis estadísticos complejos y visualizar resultados de manera clara y comprensible.

La programación también permite la automatización de tareas repetitivas, lo que ahorra tiempo y reduce errores humanos. Esto es particularmente útil en la ecología, donde la recolección y el procesamiento de datos pueden ser tareas laboriosas.

Los lenguajes de programación facilitan el desarrollo de modelos y simulaciones que pueden predecir comportamientos ecológicos y biológicos bajo diferentes escenarios. Estas herramientas son cruciales para entender fenómenos complejos y tomar decisiones informadas.

La programación permite a los biólogos y ecólogos colaborar más efectivamente con científicos de otras disciplinas, como matemáticos, estadísticos y expertos en informática. Esto fomenta un enfoque interdisciplinario que puede conducir a descubrimientos innovadores.

En la ciencia moderna, la reproducibilidad es clave. La programación asegura que los análisis puedan ser replicados y verificados por otros científicos, lo cual es fundamental para la validez de los resultados.

El conocimiento de lenguajes de programación abre la puerta al uso de recursos avanzados, como bases de datos genómicas, herramientas de bioinformática y plataformas de modelado ecológico. Esto amplía las capacidades de investigación y desarrollo en estos campos.

Conocer un lenguaje de programación no solo mejora las habilidades técnicas de los científicos, sino que también fortalece su proceso de razonamiento. La capacidad de escribir código y crear algoritmos fomenta un pensamiento lógico y estructurado, lo cual es esencial para el análisis científico riguroso.

Tercer parcial programación con lenguaje Python

https://jestrada2020.github.io/EvTercerParcial/

Instalación de Python

1. **Primer paso: Descarga de Python** Ir al sitio web oficial de Python en https://www.python.org/. En la página de inicio, deberías ver un botón grande que dice ”Download Python”. Haz clic en él.

2. **Segundo paso: Seleccionar la versión de Python** En la página de descarga, verás varias versiones de Python disponibles. Selecciona la versión que prefieras. Si no estás seguro, la última versión estable es una buena elección.

3. **Tercer paso: Descarga del instalador** Desplázate hacia abajo en la página de descarga hasta llegar a la sección ”Files”. Aquí, encontrarás los instaladores para diferentes sistemas operativos.

   Para Windows, deberías ver un conjunto de instaladores. Si tu sistema operativo es de 64 bits, descarga el instalador que termine en ”Windows x86-64 executable installer”. Si tu sistema es de 32 bits, descarga el instalador que termine en ”Windows x86 executable installer”.

4. **Cuarto paso: Ejecutar el instalador** Una vez que se complete la descarga, haz doble clic en el archivo .exe que acabas de descargar.

   Windows puede mostrar una advertencia de seguridad. Si es así, haz clic en “Ejecutar” o “Allow” para permitir que el instalador se ejecute.

5. **Quinto paso: Configuración de la instalación** En la primera pantalla del instalador, asegúrate de marcar la opción “Add Python x.x to PATH”. Esto agregará automáticamente Python al PATH de tu sistema, lo que te permitirá ejecutar comandos Python desde cualquier ubicación en tu sistema. Luego, haz clic en “Install Now” para comenzar la instalación.

6. **Sexto paso: Instalación** El instalador comenzará a instalar Python en tu sistema. Esto puede llevar unos minutos. Una vez que la instalación esté completa, verás una pantalla que dice “Setup was successful”. Marca la opción “Disable path length limit” si aparece y luego haz clic en “Close” para finalizar la instalación.

7. **Séptimo paso: Verificar la instalación** Abre el símbolo del sistema (cmd) o PowerShell y escr

```

# python
2 + 3
5
10 - 4
6
5 * 6
30
20 / 4
5.0

python --version. Deberias ver la version de Python que acabas de instalar.
```

Listing 1: Código de ejemplo en Python

0.1. Ejecución de un script de Python

1. **Creación del script:** Primero, se escribe el código Python en un archivo de texto. Este archivo puede tener cualquier extensión, pero comúnmente se utiliza la extensión ”.py”para indicar que es un script de Python.

2. **Interpretación del código:** Cuando ejecutas el script, el intérprete de Python lee y analiza el contenido del archivo línea por línea. El intérprete convierte las instrucciones escritas en código Python en instrucciones que la computadora puede entender y ejecutar.

3. **Ejecución del código:** Una vez que el intérprete ha analizado todo el código, comienza a ejecutar las instrucciones en el orden en que aparecen en el archivo. Esto implica realizar cálculos, mostrar resultados en la pantalla, interactuar con archivos, bases de datos u otros programas, entre otras acciones, según lo que esté especificado en el código.

4. **Resultados:** Durante la ejecución, el script puede producir resultados como mensajes de texto, números, gráficos, archivos creados o modificados, etc. Estos resultados pueden mostrarse en la consola, guardarse en archivos o ser utilizados para realizar otras acciones.

5. **Finalización de la ejecución:** Una vez que todas las instrucciones del script han sido ejecutadas o si ocurre algún error que detiene la ejecución, el script finaliza su ejecución. En este punto, el programa termina y cualquier recurso utilizado por el script (como memoria o conexiones de red) se libera.

0.2. Uso de la línea de comandos de Python (REPL - Read-Eval-Print Loop)

1. Calculadora simple: Puedes usar el REPL de Python como una calculadora básica. Por ejemplo, escribe python en tu terminal para iniciar el REPL y luego puedes realizar operaciones matemáticas como suma, resta, multiplicación y división:

```
# python
2 + 3
5
10 - 4
6
5 * 6
30
20 / 4
5.0
```

Listing 2: Código de ejemplo en Python

2. Manipulación de cadenas de texto: Puedes realizar operaciones y manipulaciones de cadenas de texto directamente en el REPL. Por ejemplo:

```
# python
saludo = ‘‘Hola, mundo!’’
print(saludo)
Hola, mundo!
saludo.upper()
’HOLA, MUNDO!’
saludo.split(’,’)
[’Hola’, ’ mundo!’]
```

Listing 3: Código de ejemplo en Python

3. Experimentación con funciones: Puedes probar y experimentar con funciones y definiciones directamente en el REPL. Por ejemplo:

```
# python
def cuadrado(numero):
...     return numero ** 2
...
cuadrado(5)
25
cuadrado(8)
64
```

Listing 4: Código de ejemplo en Python

4. Exploración de bibliotecas: Puedes importar bibliotecas y explorar sus funciones y métodos directamente en el REPL. Por ejemplo, importemos la biblioteca math y utilicemos algunas de sus funciones:

```
#python
import math
math.sqrt(25)
5.0
math.factorial(5)
120
```

Listing 5: Código de ejemplo en Python

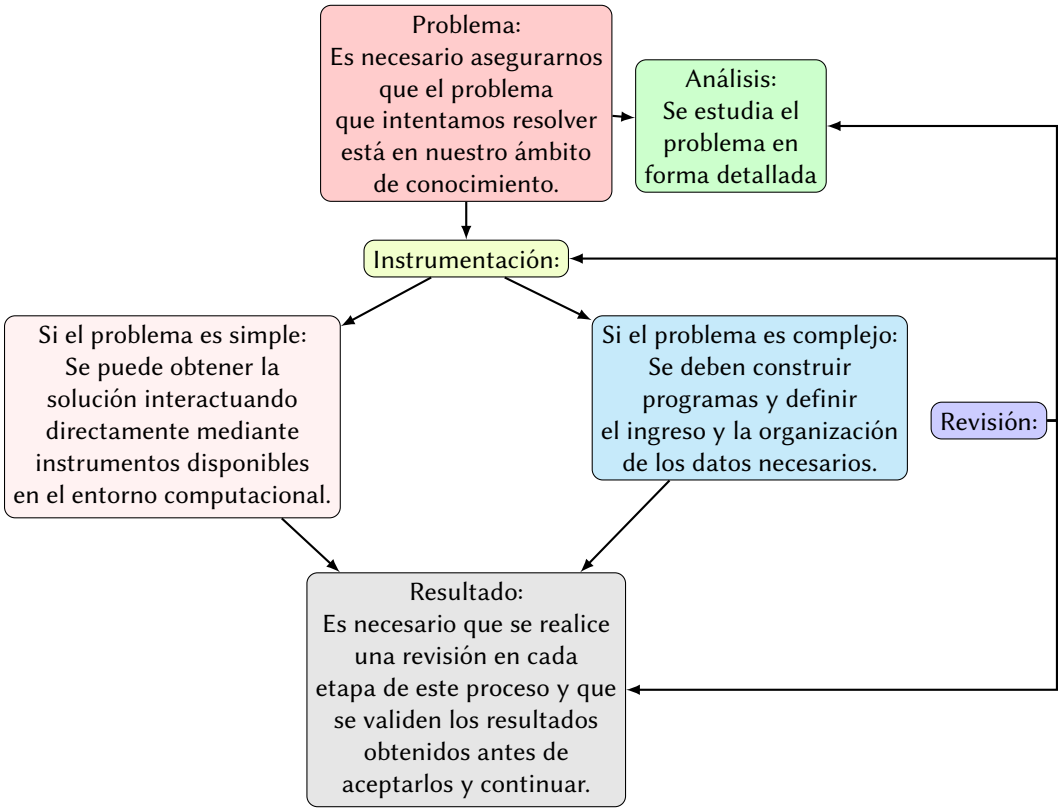

Figura 1: Esquema para resolver problemas usando el computador

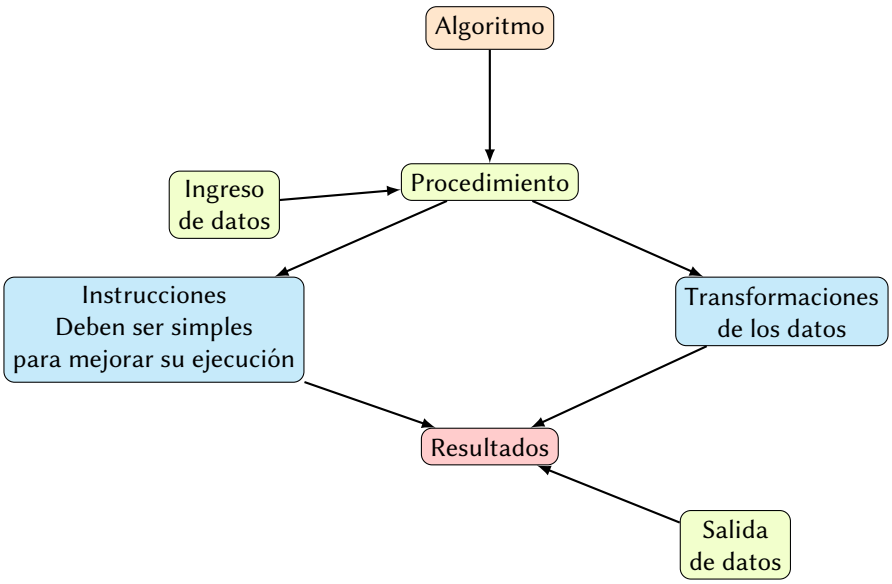

Figura 2: Esquema de un algoritmo

Tipos de datos y variables

El lenguaje de programación Python es conocido por su simplicidad y legibilidad, lo que lo convierte en una excelente opción para principiantes. Sin embargo, para aprovechar al máximo sus capacidades, es crucial entender los tipos de datos y las variables. Estos conceptos forman la base sobre la cual se construye todo programa, afectando tanto el rendimiento como la manejabilidad del código.

Las variables son esenciales para almacenar y manipular datos. En Python, la asignación de variables es simple y dinámica, permitiendo a los programadores centrarse más en la lógica del problema que en la gestión de tipos de datos.

Python soporta varios tipos de datos numéricos, incluyendo enteros (**`int`**) y números de punto flotante (**`float`**). Comprender cuándo y cómo usar estos tipos es esencial para realizar cálculos precisos y eficientes.

```
# Ejemplo de tipos numericos
a = 10      # Entero
b = 3.14    # Punto flotante
c = a + b
print(c)    # Salida: 13.14
```

Listing 6: Código de ejemplo de tipos numéricos

Las cadenas de texto (**`str`**) son utilizadas para manejar texto en Python. Saber cómo manipular cadenas es vital para tareas como el procesamiento de datos y la generación de reportes.

```
# Ejemplo de cadena de texto
nombre = "Juan"
saludo = "Hola, " + nombre
print(saludo)    # Salida: Hola, Juan
```

Listing 7: Código de ejemplo de cadenas de texto

0.3. Variables en Python

Las variables son esenciales para almacenar y manipular datos. En Python, la asignación de variables es simple y dinámica, permitiendo a los programadores centrarse más en la lógica del problema que en la gestión de tipos de datos.

```
# Ejemplo de variables
x = 10
y = "Hola"
z = 3.14
print(x, y, z)    # Salida: 10 Hola 3.14
```

Listing 8: Código de ejemplo de variables

0.4. Importancia de Conocer Tipos de Datos y Variables

0.4.1. Eficiencia y Rendimiento

El conocimiento adecuado de los tipos de datos permite a los programadores escribir código más eficiente y optimizado. Por ejemplo, usar listas en lugar de cadenas de texto para manipular secuencias puede mejorar significativamente el rendimiento.

0.4.2. Claridad y Mantenibilidad

Comprender cómo y cuándo usar diferentes tipos de datos y variables mejora la claridad del código, haciéndolo más fácil de leer y mantener. Esto es crucial para proyectos a largo plazo y colaborativos.

0.4.3. Detección y Prevención de Errores

Un buen manejo de los tipos de datos ayuda a prevenir errores comunes, como intentar realizar operaciones matemáticas en cadenas de texto. La gestión adecuada de variables y tipos de datos puede reducir significativamente los errores en el código.

0.5. Números (enteros, flotantes, complejos)

1. Enteros (**`str`**): Los números enteros en Python son números sin parte fraccionaria, positivos o negativos, y no tienen límite de tamaño en Python 3.x. Se representan mediante el tipo de datos int.

```
# Ejemplo de numeros enteros
numenteropositivo = 10
numenteronegativo = -5
```

Listing 9: Código de ejemplo en Python

2. Flotantes (**`float`**): Los números de punto flotante en Python son números que tienen una parte fraccionaria, representados mediante el tipo de datos float. Pueden representar números reales y tienen una precisión limitada.

```
# Ejemplo de numeros flotantes
numflotante = 3.14
otroflotante = 2.5e-3  # Notacion cientifica: 2.5 x 10--3
```

Listing 10: Código de ejemplo en Python

3. Complejos (complex): Los números complejos en Python tienen una parte real y una parte imaginaria, representados mediante el tipo de datos complex. Se expresan utilizando la unidad imaginaria j.

```
# Ejemplo de numeros complejos
numcomplejo = 2 + 3j
otrocomplejo = 1 - 2j
```

Listing 11: Código de ejemplo en Python

0.6. Cadenas de caracteres (strings)

Las cadenas de caracteres, o strings en inglés, son una secuencia de caracteres que representan texto. En Python, se pueden crear utilizando comillas simples (’), comillas dobles (“) o incluso comillas triples (”’ o “””).

1. Creación de cadenas: Puedes crear cadenas utilizando comillas simples o dobles:

```
cadenasimple = 'Hola, mundo!'
cadenadoble = ``Hola, mundo''
```

Listing 12: Código de ejemplo en Python

También puedes usar comillas triples para cadenas multilinea:

```
cadenamultilinea = '''Este es un ejemplo de una cadena multilinea.'''
```

Listing 13: Código de ejemplo en Python

2. Acceso a caracteres: Puedes acceder a caracteres individuales dentro de una cadena utilizando corchetes y un índice:

```
cadena = ``Python''
primercaracter = cadena[0]  # 'P'
tercercaracter = cadena[2]  # 't'
```

Listing 14: Código de ejemplo en Python

3. Concatenación de cadenas: Puedes combinar o concatenar cadenas utilizando el operador de suma (+):

```
saludo = ``Hola''
nombre = ``Juan''
mensaje = saludo + ``, '' + nombre + ``!''
```

Listing 15: Código de ejemplo en Python

4. Métodos de cadena: Python ofrece una variedad de métodos incorporados para manipular cadenas, como convertir a mayúsculas, minúsculas, dividir, reemplazar, etc. Aquí hay algunos ejemplos:

```
cadena = ``python es genial''
mayusculas = cadena.upper()  # 'PYTHON ES GENIAL'
minusculas = cadena.lower()  # 'python es genial'
palabras = cadena.split()    # ['python', 'es', 'genial']
reemplazada = cadena.replace('genial', 'increible')
# 'python es increible'
```

Listing 16: Código de ejemplo en Python

5. Formato de cadenas: Puedes formatear cadenas utilizando el método .format() o f-strings (disponibles a partir de Python 3.6):

```
nombre = "Maria"
edad = 30
mensaje = "Hola, mi nombre es - y tengo - anos.".format(nombre, edad)
mensajefstring = f"Hola, mi nombre es -nombre y tengo -edad anos."
```

Listing 17: Código de ejemplo en Python

0.7. Listas

0.7.1. Introducción

Las listas en Python son estructuras de datos que permiten almacenar múltiples elementos en una sola variable. Su capacidad para contener elementos heterogéneos y su facilidad de manipulación las hacen indispensables en diversas aplicaciones científicas. En biología y ecología, las listas facilitan el manejo y análisis de datos complejos y voluminosos.

0.7.2. Importancia de las Listas

0.7.3. Flexibilidad

Las listas pueden contener cualquier tipo de dato, incluidos números, cadenas, y otros objetos. Esta flexibilidad permite a los científicos almacenar y procesar diferentes tipos de información de manera coherente.

0.7.4. Facilidad de Manipulación

Las listas en Python son dinámicas, lo que significa que pueden crecer y reducirse en tamaño según sea necesario. Además, Python proporciona una amplia gama de métodos incorporados para manipular listas de manera eficiente.

0.7.5. Compatibilidad con Herramientas Científicas

Muchas bibliotecas científicas en Python, como NumPy y pandas, están diseñadas para trabajar de manera óptima con listas y estructuras de datos similares. Esto permite una integración fluida de las listas con herramientas avanzadas de análisis y visualización de datos.

0.8. Ejemplos Prácticos en Biología y Ecología

0.8.1. Almacenamiento de Secuencias Genéticas

En biología molecular, las listas pueden utilizarse para almacenar secuencias de nucleótidos o aminoácidos. Este almacenamiento es fundamental para realizar análisis de secuencias, como la búsqueda de patrones o la comparación entre diferentes secuencias.

```
# Ejemplo de secuencia genetica almacenada en una lista
secuenciaadn = ['A', 'T', 'G', 'C', 'A', 'A', 'T', 'G', 'C', 'C']
print(secuenciaadn)
```

Listing 18: Almacenamiento de secuencias genéticas

0.8.2. Registro de Datos de Campo

En ecología, las listas pueden utilizarse para registrar datos de campo, como la observación de especies, condiciones ambientales y coordenadas geográficas. Este tipo de datos es esencial para estudios de biodiversidad y monitoreo ambiental.

```
# Ejemplo de registro de datos de campo
observaciones = [
-"especie": "Puma concolor", "ubicacion": "Parque Nacional", "cantidad": 2 ,
-"especie": "Quercus robur", "ubicacion": "Bosque", "cantidad": 15
]
print(observaciones)
```

Listing 19: Registro de datos de campo

0.8.3. Modelado de Poblaciones

Las listas son útiles para modelar y simular el crecimiento de poblaciones a lo largo del tiempo. Esto permite a los ecólogos predecir tendencias poblacionales y evaluar el impacto de diferentes factores ambientales.

```
# Ejemplo de modelado de poblacion a lo largo del tiempo
poblacion = [100, 150, 200, 250, 300, 350]
crecimientoanual = 50
for year in range(6, 11):
nuevapoblacion = poblacion[-1] + crecimientoanual
poblacion.append(nuevapoblacion)
print(poblacion)
```

Listing 20: Modelado de poblaciones

0.8.4. Análisis de Datos Experimentales

En estudios experimentales, las listas permiten almacenar y analizar datos obtenidos en diferentes condiciones experimentales. Esto facilita la comparación y el análisis estadístico de los resultados.

```
# Ejemplo de almacenamiento de datos experimentales
resultadosexperimento = [
-"tratamiento": "Control", "medida": 5.6,
-"tratamiento": "Tratamiento A", "medida": 7.8,
-"tratamiento": "Tratamiento B", "medida": 6.4
]
print(resultadosexperimento)
```

Listing 21: Análisis de datos experimentales

0.9. Tuplas

0.9.1. Introducción

En Python, las tuplas son estructuras de datos que permiten almacenar múltiples elementos en una sola variable de manera inmutable. Esta inmutabilidad ofrece ventajas en términos de integridad y eficiencia, haciendo que las tuplas sean útiles en diversas aplicaciones científicas, incluyendo biología y ecología.

0.9.2. Importancia de las Tuplas

0.9.3. Inmutabilidad

La inmutabilidad de las tuplas asegura que los datos almacenados no se modifiquen accidentalmente durante la ejecución del programa. Esto es particularmente útil en situaciones donde la integridad de los datos es crucial.

0.9.4. Eficiencia

Las tuplas son más eficientes en términos de uso de memoria y velocidad de ejecución en comparación con las listas. Esto las hace ideales para almacenar conjuntos de datos grandes y realizar operaciones rápidas.

0.9.5. Orden y Estructura

Las tuplas mantienen el orden de los elementos, lo que es esencial para tareas donde la secuencia de datos es importante. Además, su estructura simple facilita su uso en combinación con otras estructuras de datos y funciones.

0.10. Ejemplos Prácticos en Biología y Ecología

0.10.1. Coordenadas Geográficas

En estudios de campo, las tuplas pueden utilizarse para almacenar coordenadas geográficas de ubicaciones de observación. Esto asegura que las coordenadas permanezcan consistentes y no se modifiquen accidentalmente.

```
# Ejemplo de coordenadas geograficas
ubicacion = (34.0522, -118.2437)
print(ubicacion)
```

Listing 22: Coordenadas geográficas almacenadas en tuplas

0.10.2. Datos de Secuencias Genéticas

Las tuplas son ideales para almacenar pares de bases nucleotídicas o secuencias de aminoácidos, manteniendo la integridad de los datos genéticos y permitiendo análisis precisos.

```
# Ejemplo de secuencia genetica
secuenciaadn = ('A', 'T', 'G', 'C', 'A', 'A', 'T', 'G', 'C', 'C')
print(secuenciaadn)
```

Listing 23: Secuencias genéticas almacenadas en tuplas

0.10.3. Parámetros de Modelos Ecológicos

En la modelación ecológica, las tuplas pueden usarse para almacenar conjuntos de parámetros que deben permanecer constantes durante las simulaciones, como tasas de crecimiento y mortalidad.

```
# Ejemplo de parametros de modelo ecologico
parametros = (0.1, 0.05, 100)
tasacrecimiento, tasamortalidad, capacidadcarga = parametros
print(parametros)
```

Listing 24: Parámetros de modelos ecológicos

0.10.4. Resultados de Experimentos

Las tuplas pueden utilizarse para almacenar los resultados de experimentos, asegurando que los datos permanezcan inalterados y listos para análisis posteriores.

```
# Ejemplo de resultados de experimentos
resultadoexperimento = ("Tratamiento A", 7.8)
print(resultadoexperimento)
```

Listing 25: Resultados de experimentos almacenados en tuplas

0.11. Diccionarios

0.11.1. Introducción

En Python, los diccionarios son estructuras de datos que permiten almacenar datos en pares clave-valor. Esta capacidad de asociar claves únicas con valores correspondientes es extremadamente útil en diversas aplicaciones científicas, especialmente en biología y ecología, donde la organización y el acceso eficiente a grandes volúmenes de datos son esenciales.

0.11.2. Importancia de los Diccionarios

0.11.3. Acceso Rápido y Eficiente

Los diccionarios permiten un acceso rápido y eficiente a los datos, ya que las operaciones de búsqueda, inserción y eliminación tienen un tiempo promedio de complejidad O(1). Esto es crucial cuando se trabaja con grandes conjuntos de datos.

0.11.4. Flexibilidad

Los diccionarios en Python pueden almacenar cualquier tipo de dato, incluyendo otros diccionarios. Esta flexibilidad permite una estructura de datos altamente anidada y compleja, adecuada para las necesidades de los científicos.

0.11.5. Claridad y Organización

El uso de claves descriptivas en los diccionarios mejora la claridad y organización del código, haciendo que los datos sean más legibles y manejables.

0.12. Ejemplos Prácticos en Biología y Ecología

0.12.1. Anotaciones Genéticas

En biología molecular, los diccionarios pueden utilizarse para almacenar anotaciones genéticas, donde las claves son los nombres de los genes y los valores son sus respectivas funciones o características.

```
# Ejemplo de anotaciones geneticas
anotacionesgeneticas = -
  "BRCA1": "Gen asociado con el cancer de mama",
  "TP53": "Gen supresor de tumores",
  "MYH7": "Gen asociado con miocardiopatias"

print(anotacionesgeneticas)
```

Listing 26: Anotaciones genéticas en un diccionario

0.12.2. Registros de Observación de Especies

En ecología, los diccionarios pueden utilizarse para registrar observaciones de especies, donde las claves son los nombres de las especies y los valores son detalles de la observación como la ubicación y la cantidad.

```
# Ejemplo de registros de observacion de especies
observacionesespecies = -
  "Puma concolor": -"ubicacion": "Parque Nacional", "cantidad": 2 ,
  "Quercus robur": -"ubicacion": "Bosque", "cantidad": 15

print(observacionesespecies)
```

Listing 27: Registros de observación de especies en un diccionario

0.12.3. Modelado de Ecosistemas

En la modelación ecológica, los diccionarios pueden usarse para almacenar parámetros de modelos que describen las interacciones entre diferentes especies dentro de un ecosistema.

```
# Ejemplo de modelado de ecosistemas
modeloecosistema = -
  "depredador": -"especie": "Lobo", "tasapredacion": 0.05,
  "presa": -"especie": "Ciervo", "tasareproduccion": 0.1

print(modeloecosistema)
```

Listing 28: Modelado de ecosistemas usando diccionarios

0.12.4. Análisis de Datos Ambientales

En estudios ambientales, los diccionarios pueden utilizarse para organizar datos de sensores ambientales, donde las claves son los tipos de sensores y los valores son las lecturas correspondientes.

```
# Ejemplo de analisis de datos ambientales
datosambientales = -
  "temperatura": [22.4, 23.1, 21.8, 22.0],
  "humedad": [60, 65, 58, 63],
  "nivelCO2": [400, 420, 390, 405]

print(datosambientales)
```

Listing 29: Análisis de datos ambientales con diccionarios

0.13. Conjuntos

0.13.1. Introducción

En Python, los conjuntos son estructuras de datos que permiten almacenar colecciones de elementos únicos sin un orden particular. Esta capacidad para manejar datos únicos y realizar operaciones matemáticas como la unión, la intersección y la diferencia, es extremadamente útil en diversas aplicaciones científicas, especialmente en biología y ecología.

0.13.2. Importancia de los Conjuntos

0.13.3. Unicidad de Elementos

Los conjuntos en Python garantizan que cada elemento sea único, lo que es crucial para aplicaciones donde la duplicación de datos podría conducir a resultados incorrectos o ineficientes.

0.13.4. Operaciones Matemáticas

Los conjuntos permiten realizar operaciones matemáticas como la unión, la intersección y la diferencia de manera eficiente. Estas operaciones son útiles para comparar y combinar datos de diferentes fuentes.

0.13.5. Eficiencia

Las operaciones de búsqueda, inserción y eliminación en un conjunto tienen un tiempo promedio de complejidad O(1). Esto hace que los conjuntos sean ideales para manejar grandes volúmenes de datos donde la eficiencia es clave.

0.14. Ejemplos Prácticos en Biología y Ecología

0.14.1. Análisis de Diversidad de Especies

En ecología, los conjuntos pueden utilizarse para analizar la diversidad de especies en diferentes hábitats. Los conjuntos permiten comparar las especies observadas en distintas áreas y determinar la diversidad y la similitud entre ellas.

```
# Ejemplo de analisis de diversidad de especies
especieshabitat1 = -"Puma concolor", "Quercus robur", "Canis lupus"
especieshabitat2 = -"Canis lupus", "Panthera onca", "Quercus robur"

# Especies unicas en ambos habitats
especiesunicas = especieshabitat1.union(especieshabitat2)
print(especiesunicas)

# Especies comunes en ambos habitats
especiescomunes = especieshabitat1.intersection(especieshabitat2)
print(especiescomunes)
```

Listing 30: Análisis de diversidad de especies

0.14.2. Comparación de Secuencias Genéticas

En biología molecular, los conjuntos pueden utilizarse para comparar secuencias genéticas y encontrar elementos comunes o únicos entre diferentes secuencias.

```
# Ejemplo de comparacion de secuencias geneticas
secuencia1 = -"A", "T", "G", "C"
secuencia2 = -"G", "C", "A", "T", "A"

# Elementos comunes entre ambas secuencias
comunes = secuencia1.intersection(secuencia2)
print(comunes)

# Elementos unicos en cada secuencia
```

```
unicos = secuencia1.symmetricdifference(secuencia2)
print(unicos)
```

Listing 31: Comparación de secuencias genéticas

0.14.3. Filtrado de Datos Experimentales

En estudios experimentales, los conjuntos pueden utilizarse para filtrar datos y eliminar duplicados, asegurando que cada observación o medición se considere una sola vez.

```
# Ejemplo de filtrado de datos experimentales
mediciones = [1.2, 2.3, 2.3, 4.5, 1.2, 5.6]
medicionesunicas = set(mediciones)
print(medicionesunicas)
```

Listing 32: Filtrado de datos experimentales

0.14.4. Gestión de Datos de Campo

En ecología, los conjuntos pueden utilizarse para gestionar datos de campo, como la observación de especies en diferentes puntos de muestreo, permitiendo una organización y análisis eficiente de los datos recogidos.

```
# Ejemplo de gestion de datos de campo
observacionespunto1 = -"Especie A", "Especie B", "Especie C"
observacionespunto2 = -"Especie B", "Especie C", "Especie D"

# Especies observadas en al menos un punto
especiestotales = observacionespunto1.union(observacionespunto2)
print(especiestotales)

# Especies observadas en ambos puntos
especiescomunes = observacionespunto1.intersection(observacionespunto2)
print(especiescomunes)
```

Listing 33: Gestión de datos de campo

Operadores

0.14.5. Introducción

En Python, los operadores son símbolos especiales que llevan a cabo operaciones aritméticas, lógicas, de comparación y más. Son esenciales para el manejo y la transformación de datos, lo que los convierte en herramientas indispensables en aplicaciones científicas como biología y ecología.

0.14.6. Importancia de los Operadores

0.14.7. Operaciones Aritméticas

Los operadores aritméticos permiten realizar cálculos matemáticos fundamentales, que son cruciales en el análisis de datos y la modelización científica.

0.14.8. Comparaciones

Los operadores de comparación son esenciales para la toma de decisiones y el control de flujo en los programas, permitiendo la evaluación de condiciones y la ejecución de código basado en esas evaluaciones.

0.14.9. Operaciones Lógicas

Los operadores lógicos permiten combinar múltiples condiciones y realizar evaluaciones complejas, lo cual es fundamental en la programación de algoritmos avanzados y análisis de datos.

0.14.10. Asignación y Manipulación de Datos

Los operadores de asignación y manipulación de datos facilitan la actualización y modificación de variables, permitiendo la gestión dinámica de datos a lo largo del programa.

0.15. Ejemplos Prácticos en Biología y Ecología

0.15.1. Cálculo de Crecimiento Poblacional

En ecología, los operadores aritméticos pueden utilizarse para calcular el crecimiento poblacional de una especie a lo largo del tiempo.

```
# Ejemplo de calculo de crecimiento poblacional
poblacioninicial = 100
tasacrecimiento = 0.05
tiempo = 10  # anos

poblacionfinal = poblacioninicial * (1 + tasacrecimiento) ** tiempo
print(poblacionfinal)
```

Listing 34: Cálculo de crecimiento poblacional

0.15.2. Comparación de Datos Genéticos

En biología molecular, los operadores de comparación pueden utilizarse para identificar secuencias genéticas que cumplen con ciertos criterios.

```
# Ejemplo de comparacion de datos geneticos
secuenciaadn = ”ATGCGTACG”
motivo = ”CGT”

if motivo in secuenciaadn:
print(”Motivo encontrado”)
else:
print(”Motivo no encontrado”)
```

Listing 35: Comparación de datos genéticos

0.15.3. Análisis de Datos Ambientales

Los operadores lógicos pueden utilizarse para filtrar y analizar datos ambientales, como la temperatura y la humedad, para identificar condiciones específicas.

```
# Ejemplo de analisis de datos ambientales
temperatura = 22.5
humedad = 60

if temperatura ¿ 20 and humedad ¡ 65:
print(”Condiciones ideales para el estudio”)
else:
print(”Condiciones no ideales”)
```

Listing 36: Análisis de datos ambientales

0.15.4. Modelado de Interacciones Ecológicas

En la modelación ecológica, los operadores de asignación y manipulación de datos pueden utilizarse para simular interacciones entre especies.

```
# Ejemplo de modelado de interacciones ecologicas
depredadores = 20
presas = 100
tasapredacion = 0.1

for   in range(5):  # 5 ciclos de simulacion
presas -= depredadores * tasapredacion
if presas ¡ 0:
presas = 0
print(f”Depredadores: -depredadores , Presas: -presas ”)
```

Listing 37: Modelado de interacciones ecológicas

Control de flujo

0.15.5. Introducción

El control de flujo en Python permite a los programadores dirigir la ejecución de su código basado en condiciones y repeticiones. Esto es fundamental para la toma de decisiones, la ejecución de tareas repetitivas y el manejo de datos dinámicos en diversas aplicaciones científicas, incluyendo biología y ecología.

0.15.6. Importancia del Control de Flujo

0.15.7. Toma de Decisiones

El control de flujo permite a los programas tomar decisiones basadas en condiciones específicas, lo cual es crucial para responder a diferentes situaciones y datos.

0.15.8. Repetición de Tareas

Las estructuras de bucle permiten la repetición de tareas múltiples veces, lo que es esencial para el procesamiento de grandes volúmenes de datos y la ejecución de simulaciones.

0.15.9. Ejecución Condicional

La ejecución condicional permite que diferentes bloques de código se ejecuten en función de los datos de entrada, mejorando la flexibilidad y la capacidad de respuesta de los programas.

0.15.10. Manejo de Errores

El control de flujo también facilita el manejo de errores y excepciones, lo cual es importante para garantizar que los programas se comporten de manera robusta frente a datos inesperados o incorrectos.

0.16. Ejemplos Prácticos en Biología y Ecología

0.16.1. Análisis de Datos de Campo

En ecología, el control de flujo puede utilizarse para analizar datos de campo, tomando decisiones basadas en las condiciones observadas.

```
# Ejemplo de analisis de datos de campo
observaciones = [-”especie”: ”Puma concolor”, ”cantidad”: 2 ,
-”especie”: ”Quercus robur”, ”cantidad”: 15 ,
-”especie”: ”Canis lupus”, ”cantidad”: 3 ]

for observacion in observaciones:
if observacion[”cantidad”] ¿ 10:
print(f”Alta abundancia de -observacion[’especie ’] ”)
else:
print(f”Baja abundancia de -observacion[’especie ’] ”)
```

Listing 38: Análisis de datos de campo

0.16.2. Modelado de Poblaciones

El control de flujo puede utilizarse para modelar el crecimiento de poblaciones bajo diferentes condiciones, ejecutando simulaciones basadas en tasas de crecimiento y otros parámetros.

```
# Ejemplo de modelado de poblaciones
poblacion = 100
tasacrecimiento = 0.1

for year in range(1, 11):
poblacion *= (1 + tasacrecimiento)
if poblacion ¿ 200:
print(f"Poblacion en ano -year  ha excedido 200 individuos.")
break
print(f"Ano -year :  Poblacion = -poblacion ")
```

Listing 39: Modelado de poblaciones

0.16.3. Procesamiento de Secuencias Genéticas

En biología molecular, el control de flujo permite el procesamiento y análisis de secuencias genéticas, identificando patrones y motivos específicos.

```
# Ejemplo de procesamiento de secuencias geneticas
secuenciaadn = "ATGCGTACGTTAGCTAGC"

for base in secuenciaadn:
if base == "A":
print("Adenina encontrada")
elif base == "T":
print("Timina encontrada")
elif base == "G":
print("Guanina encontrada")
elif base == "C":
print("Citosina encontrada")
```

Listing 40: Procesamiento de secuencias genéticas

0.16.4. Simulación de Ecosistemas

El control de flujo es esencial para simular interacciones en un ecosistema, permitiendo la ejecución de bucles que representan el paso del tiempo y las interacciones entre especies.

```
# Ejemplo de simulacion de ecosistemas
depredadores = 10
presas = 50
tasapredacion = 0.1
tasareproduccion = 0.05

for month in range(1, 13):
presas -= depredadores * tasapredacion * presas
presas += presas * tasareproduccion
if presas menor o igual 0:
presas = 0
print(f"Mes -month :  Todas las presas han sido depredadas.")
break
print(f"Mes -month :  Depredadores = -depredadores ,  Presas = -presas ")
```

Listing 41: Simulación de ecosistemas

Funciones

0.16.5. Introducción

En Python, las funciones son bloques de código reutilizables que realizan tareas específicas. Permiten la organización y modularización del código, mejorando su legibilidad y mantenibilidad. Las funciones son especialmente útiles en aplicaciones científicas, como biología y ecología, donde el análisis y la manipulación de datos son fundamentales.

0.16.6. Importancia de las Funciones

0.16.7. Modularización

Las funciones permiten dividir el código en módulos más pequeños y manejables. Esto facilita la comprensión y el mantenimiento del código, especialmente en proyectos grandes y complejos.

0.16.8. Reutilización

Las funciones pueden reutilizarse en diferentes partes del programa, reduciendo la redundancia y el esfuerzo de programación. Esto es crucial para la eficiencia y la productividad.

0.16.9. Abstracción

Las funciones permiten abstraer detalles específicos de la implementación, proporcionando una interfaz clara y sencilla para realizar tareas complejas. Esto mejora la claridad y la legibilidad del código.

0.16.10. Mantenibilidad

Las funciones facilitan la actualización y modificación del código. Cambios en la lógica de una función no afectan otras partes del programa, siempre y cuando la interfaz de la función permanezca consistente.

0.17. Ejemplos Prácticos en Biología y Ecología

0.17.1. Cálculo de Índices de Biodiversidad

En ecología, las funciones pueden utilizarse para calcular índices de biodiversidad, como el índice de Shannon, a partir de datos de abundancia de especies.

```
# Funcion para calcular el indice de Shannon
import math

def indiceshannon(abundancias):
total = sum(abundancias)
shannon = 0.0
for abundancia in abundancias:
proporcion = abundancia / total
shannon -= proporcion * math.log(proporcion)
return shannon

# Ejemplo de uso
abundancias = [10, 20, 30, 40]
```

```
print(f"Indice de Shannon:-indiceshannon(abundancias) ")
```

Listing 42: Cálculo de índices de biodiversidad

0.17.2. Análisis de Secuencias Genéticas

En biología molecular, las funciones pueden utilizarse para analizar secuencias genéticas, buscando patrones o motivos específicos en una secuencia de ADN.

```
# Funcion para buscar un motivo en una secuencia de ADN
def buscarmotivo(secuencia, motivo):
posiciones = []
for i in range(len(secuencia) - len(motivo) + 1):
if secuencia[i:i+len(motivo)] == motivo:
posiciones.append(i)
return posiciones

# Ejemplo de uso
secuenciaadn = "ATGCGTACGTTAGCTAGC"
motivo = "CGT"
print(f"Motivo encontrado en posiciones: -buscarmotivo(secuenciaadn, motivo) ")
```

Listing 43: Análisis de secuencias genéticas

0.17.3. Modelado de Crecimiento Poblacional

Las funciones pueden utilizarse para modelar el crecimiento poblacional de una especie bajo diferentes condiciones ambientales y parámetros de crecimiento.

```
# Funcion para modelar el crecimiento poblacional
def crecimientopoblacional(poblacioninicial, tasacrecimiento, tiempo):
poblacion = poblacioninicial
for   in range(tiempo):
poblacion *= (1 + tasacrecimiento)
return poblacion

# Ejemplo de uso
poblacioninicial = 100
tasacrecimiento = 0.05
tiempo = 10  # anos
print(f"Poblacion despues de -tiempo  anos: -crecimientopoblacional(poblacioninicial,
  tasacrecimiento, tiempo) ")
```

Listing 44: Modelado de crecimiento poblacional

0.17.4. Simulación de Interacciones Ecológicas

Las funciones pueden utilizarse para simular interacciones entre diferentes especies en un ecosistema, como la depredación y la competencia.

```
# Funcion para simular interacciones ecologicas
def interaccionesecologicas(depredadores, presas, tasapredacion, tasareproduccion, tiempo):
for   in range(tiempo):
presas -= depredadores * tasapredacion * presas
presas += presas * tasareproduccion
if presas menor o igual 0:
presas = 0
break
return presas

# Ejemplo de uso
depredadores = 10
presas = 50
tasapredacion = 0.1
tasareproduccion = 0.05
tiempo = 12  # meses
```

```
17 print(f"Poblacion de presas despues de -tiempo  meses: -interaccionesecologicas(depredadores,
     presas, tasapredacion, tasareproduccion, tiempo) ")
```

Listing 45: Simulación de interacciones ecológicas

Estructuras de datos y operaciones

0.18. Introducción

En Python, las estructuras de datos como listas, tuplas, diccionarios y conjuntos, junto con las operaciones que pueden realizarse sobre ellas, son esenciales para manejar y analizar datos de manera eficiente. Estas herramientas son particularmente importantes en campos científicos como la biología y la ecología, donde el manejo de grandes conjuntos de datos es una tarea común.

0.19. Importancia de las Estructuras de Datos y Operaciones

0.19.1. Organización de Datos

Las estructuras de datos permiten organizar la información de manera lógica y accesible. Esto es crucial para la claridad y eficiencia en el análisis de datos.

0.19.2. Eficiencia Computacional

El uso adecuado de estructuras de datos y operaciones optimiza el rendimiento del código, permitiendo la manipulación y análisis de grandes volúmenes de datos de manera eficiente.

0.19.3. Flexibilidad

Python ofrece una variedad de estructuras de datos, cada una con sus propias ventajas y desventajas. Esta flexibilidad permite a los programadores elegir la herramienta adecuada para cada tarea específica.

0.19.4. Reutilización y Mantenibilidad

Las estructuras de datos y operaciones bien diseñadas facilitan la reutilización del código y mejoran su mantenibilidad, lo cual es esencial en proyectos de larga duración y colaborativos.

0.20. Ejemplos Prácticos en Biología y Ecología

0.20.1. Registro y Análisis de Observaciones de Campo

En ecología, las listas y diccionarios pueden utilizarse para registrar observaciones de campo y realizar análisis básicos sobre los datos recogidos.

```
# Ejemplo de registro y analisis de observaciones de campo
observaciones = [
-"especie": "Puma concolor", "ubicacion": "Parque Nacional", "cantidad": 2 ,
-"especie": "Quercus robur", "ubicacion": "Bosque", "cantidad": 15 ,
-"especie": "Canis lupus", "ubicacion": "Montanas", "cantidad": 3
]

# Analisis: contar el numero total de individuos observados
totalindividuos = sum(observacion["cantidad"] for observacion in observaciones)
```

```
print(f"Total de individuos observados: {totalindividuos}")
```

Listing 46: Registro y análisis de observaciones de campo

0.20.2. Comparación de Secuencias Genéticas

Las listas y conjuntos pueden utilizarse para comparar secuencias genéticas, identificando elementos comunes y únicos entre diferentes secuencias.

```
# Ejemplo de comparacion de secuencias geneticas
secuencia1 = ["A", "T", "G", "C"]
secuencia2 = ["G", "C", "A", "T", "A"]

# Usar conjuntos para encontrar elementos comunes y unicos
comunes = set(secuencia1).intersection(secuencia2)
unicos = set(secuencia1).symmetricdifference(secuencia2)

print(f"Elementos comunes: {comunes}")
print(f"Elementos unicos: {unicos}")
```

Listing 47: Comparación de secuencias genéticas

0.20.3. Modelado de Crecimiento Poblacional

Las funciones y las listas pueden combinarse para modelar el crecimiento poblacional de una especie bajo diferentes condiciones.

```
# Funcion para modelar el crecimiento poblacional
def crecimientopoblacional(poblacioninicial, tasacrecimiento, tiempo):
poblacion = poblacioninicial
poblaciones = [poblacion]
for  in range(tiempo):
poblacion *= (1 + tasacrecimiento)
poblaciones.append(poblacion)
return poblaciones

# Ejemplo de uso
poblacioninicial = 100
tasacrecimiento = 0.05
tiempo = 10  # anos
print(f"Modelo de crecimiento poblacional: {crecimientopoblacional(poblacioninicial,
  tasacrecimiento, tiempo)}")
```

Listing 48: Modelado de crecimiento poblacional

0.20.4. Simulación de Interacciones en un Ecosistema

Los diccionarios y las funciones pueden utilizarse para simular interacciones complejas entre diferentes especies en un ecosistema.

```
# Funcion para simular interacciones en un ecosistema
def interaccionesecosistema(especies, interacciones, tiempo):
for  in range(tiempo):
for especie, data in especies.items():
if especie in interacciones:
for interaccion, efecto in interacciones[especie].items():
data["cantidad"] += data["cantidad"] * efecto * especies[interaccion]["cantidad"]
return especies

# Datos iniciales
especies = {
  "depredador": {"cantidad": 10},
  "presa": {"cantidad": 50}


# Interacciones: depredacion
```

```
  interacciones = -
    "depredador": -"presa": -0.01,
    "presa": -"depredador": 0.02


  # Ejemplo de uso
  tiempo = 12  # meses
  resultado = interaccionesecosistema(especies, interacciones, tiempo)
  print(f"Simulacion de interacciones en el ecosistema: -resultado ")
```

Listing 49: Simulación de interacciones en un ecosistema

Entrada y salida de datos

0.21. Introducción

En Python, las operaciones de entrada y salida de datos permiten la interacción con el usuario y la manipulación de archivos, lo que es esencial para el procesamiento y análisis de datos. Estas operaciones son particularmente importantes en aplicaciones científicas como la biología y la ecología, donde el manejo de grandes volúmenes de datos es una tarea común.

0.22. Importancia de la Entrada y Salida de Datos

0.22.1. Interacción con el Usuario

La entrada y salida de datos permiten a los programas interactuar con los usuarios, recibiendo información y proporcionando resultados. Esto es crucial para aplicaciones que requieren datos dinámicos y la generación de reportes.

0.22.2. Almacenamiento y Recuperación de Datos

El manejo de archivos permite almacenar datos de manera persistente, facilitando su recuperación y análisis posterior. Esto es esencial para el procesamiento de grandes volúmenes de datos y la realización de análisis a largo plazo.

0.22.3. Procesamiento de Datos en Tiempo Real

Las operaciones de entrada y salida permiten el procesamiento de datos en tiempo real, lo que es crucial para aplicaciones que requieren respuestas rápidas basadas en datos dinámicos.

0.22.4. Automatización de Tareas

La capacidad de leer y escribir datos de archivos permite la automatización de tareas repetitivas, mejorando la eficiencia y reduciendo la posibilidad de errores humanos.

0.23. Ejemplos Prácticos en Biología y Ecología

0.23.1. Lectura y Análisis de Datos de Campo

En ecología, los archivos de texto pueden utilizarse para almacenar datos de campo, que luego pueden ser leídos y analizados por un programa en Python.

```
# Ejemplo de lectura de datos de campo desde un archivo
with open('datoscampo.txt', 'r') as archivo:
datos = archivo.readlines()

# Analisis: contar el numero total de individuos observados
totalindividuos = sum(int(linea.split()[2]) for linea in datos)
print(f"Total de individuos observados: -totalindividuos ")
```

Listing 50: Lectura y análisis de datos de campo

0.23.2. Escritura de Resultados de Análisis Genéticos

En biología molecular, los resultados de análisis genéticos pueden ser escritos a archivos para su posterior revisión y análisis.

```
# Ejemplo de escritura de resultados de analisis geneticos
resultados = [
-"gen": "BRCA1", "mutacion": "Si" ,
-"gen": "TP53", "mutacion": "No" ,
-"gen": "MYH7", "mutacion": "Si"
]

with open('resultadosgeneticos.txt', 'w') as archivo:
for resultado in resultados:
archivo.write(f"-resultado['gen']"t-resultado['mutacion']"n")
```

Listing 51: Escritura de resultados de análisis genéticos

0.23.3. Procesamiento de Datos de Sensores Ambientales

Los datos de sensores ambientales pueden ser leídos desde archivos CSV y procesados para análisis posteriores.

```
# Ejemplo de lectura y procesamiento de datos de sensores ambientales desde un archivo CSV
import csv

with open('datossensores.csv', 'r') as archivo:
lector = csv.reader(archivo)
cabeceras = next(lector)
datos = [fila for fila in lector]

# Analisis: calcular el promedio de temperatura
temperaturas = [float(fila[1]) for fila in datos]
promediotemperatura = sum(temperaturas) / len(temperaturas)
print(f"Promedio de temperatura: -promediotemperatura ")
```

Listing 52: Procesamiento de datos de sensores ambientales

0.23.4. Automatización de Simulaciones Ecológicas

Los resultados de simulaciones ecológicas pueden ser escritos a archivos para su posterior análisis y visualización.

```
# Ejemplo de automatizacion de simulaciones ecologicas y escritura de resultados
def simulacionecologica(poblacioninicial, tasacrecimiento, tiempo):
poblacion = poblacioninicial
poblaciones = [poblacion]
for   in range(tiempo):
poblacion *= (1 + tasacrecimiento)
poblaciones.append(poblacion)
return poblaciones

# Parametros de la simulacion
poblacioninicial = 100
tasacrecimiento = 0.05
tiempo = 10  # anos

# Ejecutar la simulacion
resultados = simulacionecologica(poblacioninicial, tasacrecimiento, tiempo)

# Escribir los resultados a un archivo
with open('resultadossimulacion.txt', 'w') as archivo:
for ano, poblacion in enumerate(resultados):
archivo.write(f"Ano -ano :  -poblacion " n")
```

Listing 53: Automatización de simulaciones ecológicas

Manejo de excepciones

0.24. Introducción

En Python, el manejo de excepciones permite gestionar errores y condiciones excepcionales de manera controlada, evitando que los programas se detengan abruptamente y asegurando su correcto funcionamiento. Esta capacidad es esencial en aplicaciones científicas como la biología y la ecología, donde el manejo adecuado de errores puede ser crucial para el análisis de datos y la toma de decisiones.

0.25. Importancia del Manejo de Excepciones

0.25.1. Robustez del Programa

El manejo de excepciones permite a los programas continuar su ejecución incluso cuando ocurren errores, asegurando que los procesos críticos no se interrumpan inesperadamente.

0.25.2. Depuración y Diagnóstico

Las excepciones proporcionan información detallada sobre la naturaleza de los errores, facilitando la depuración y el diagnóstico de problemas en el código.

0.25.3. Manejo de Errores Esperados

El manejo de excepciones permite gestionar errores esperados de manera controlada, mejorando la experiencia del usuario y la fiabilidad del programa.

0.25.4. Mantenimiento del Código

Un manejo adecuado de excepciones mejora la mantenibilidad del código, permitiendo la implementación de estrategias de recuperación y manejo de errores más eficientes.

0.26. Ejemplos Prácticos en Biología y Ecología

0.26.1. Lectura de Datos de Archivos

En ecología, los datos de campo se almacenan frecuentemente en archivos. El manejo de excepciones es crucial para asegurar que el programa pueda gestionar archivos faltantes o datos corruptos.

```
# Ejemplo de manejo de excepciones al leer un archivo de datos
try:
with open('datoscampo.txt', 'r') as archivo:
datos = archivo.readlines()
# Analisis: contar el numero total de individuos observados
totalindividuos = sum(int(linea.split()[2]) for linea in datos)
print(f"Total de individuos observados: -totalindividuos ")
except FileNotFoundError:
print("Error: El archivo de datos no se encontro.")
except ValueError:
```

```
print("Error: Datos corruptos en el archivo.")
```

Listing 54: Lectura de datos de archivos

0.26.2. Procesamiento de Datos Genéticos

En biología molecular, el manejo de excepciones es esencial para gestionar errores durante el procesamiento de secuencias genéticas, como entradas inválidas o formatos incorrectos.

```
# Ejemplo de manejo de excepciones al procesar datos geneticos
def buscarmotivo(secuencia, motivo):
try:
posiciones = []
for i in range(len(secuencia) - len(motivo) + 1):
if secuencia[i:i+len(motivo)] == motivo:
posiciones.append(i)
return posiciones
except TypeError:
print("Error: Tipo de datos invalido.")
except Exception as e:
print(f"Error inesperado: -e ")

# Ejemplo de uso
secuenciaadn = "ATGCGTACGTTAGCTAGC"
motivo = "CGT"
print(f"Motivo encontrado en posiciones: -buscarmotivo(secuenciaadn, motivo) ")
```

Listing 55: Procesamiento de datos genéticos

0.26.3. Simulación de Crecimiento Poblacional

El manejo de excepciones es importante en la simulación de crecimiento poblacional para gestionar entradas de usuario inválidas o cálculos que resulten en valores no válidos.

```
# Ejemplo de manejo de excepciones en la simulacion de crecimiento poblacional
def crecimientopoblacional(poblacioninicial, tasacrecimiento, tiempo):
try:
poblacion = poblacioninicial
for   in range(tiempo):
poblacion *= (1 + tasacrecimiento)
return poblacion
except TypeError:
print("Error: Tipo de datos invalido.")
except ValueError:
print("Error: Valor invalido para la tasa de crecimiento o el tiempo.")

# Ejemplo de uso
poblacioninicial = 100
tasacrecimiento = 0.05
tiempo = 10  # anos
print(f"Poblacion despues de -tiempo  anos: -crecimientopoblacional(poblacioninicial,
  tasacrecimiento, tiempo) ")
```

Listing 56: Simulación de crecimiento poblacional

0.26.4. Análisis de Datos Ambientales

El manejo de excepciones es crucial para gestionar errores durante el análisis de datos ambientales, como datos faltantes o formatos incorrectos en archivos CSV.

```
# Ejemplo de manejo de excepciones en el analisis de datos ambientales
import csv

try:
with open('datossensores.csv', 'r') as archivo:
lector = csv.reader(archivo)
```

```
cabeceras = next(lector)
datos = [fila for fila in lector]

# Analisis: calcular el promedio de temperatura
temperaturas = [float(fila[1]) for fila in datos]
promediotemperatura = sum(temperaturas) / len(temperaturas)
print(f"Promedio de temperatura: -promediotemperatura ")
except FileNotFoundError:
print("Error: El archivo de datos no se encontro.")
except ValueError:
print("Error: Datos corruptos en el archivo.")
except Exception as e:
print(f"Error inesperado: -e ")
```

Listing 57: Análisis de datos ambientales

Módulos y paquetes

0.27. Introducción

En Python, los módulos y paquetes son herramientas fundamentales para la organización y reutilización de código. Un módulo es un archivo que contiene definiciones y declaraciones de Python, mientras que un paquete es una colección de módulos. Estas herramientas permiten a los programadores estructurar su código de manera lógica y eficiente, facilitando la colaboración y el mantenimiento del código, especialmente en proyectos grandes y complejos.

0.28. Importancia de los Módulos y Paquetes

0.28.1. Modularización del Código

Los módulos y paquetes permiten dividir el código en bloques más pequeños y manejables, mejorando la legibilidad y facilitando la colaboración entre varios programadores.

0.28.2. Reutilización del Código

El uso de módulos y paquetes facilita la reutilización del código, permitiendo a los desarrolladores utilizar funciones y clases en múltiples proyectos sin necesidad de duplicar el código.

0.28.3. Mantenibilidad

Organizar el código en módulos y paquetes mejora la mantenibilidad, ya que los cambios en una parte del código pueden hacerse sin afectar otras partes, siempre y cuando se mantengan las interfaces de los módulos.

0.28.4. Eficiencia en el Desarrollo

El uso de módulos y paquetes acelera el desarrollo de software al permitir la integración de bibliotecas externas, ahorrando tiempo y esfuerzo en la implementación de funcionalidades comunes.

0.29. Ejemplos Prácticos en Biología y Ecología

0.29.1. Análisis de Datos Genéticos

En biología molecular, los módulos pueden utilizarse para organizar funciones relacionadas con el análisis de secuencias genéticas, facilitando la reutilización y el mantenimiento del código.

```
# Ejemplo de modulo para analisis de secuencias geneticas: secuencias.py
def buscarmotivo(secuencia, motivo):
posiciones = []
for i in range(len(secuencia) - len(motivo) + 1):
if secuencia[i:i+len(motivo)] == motivo:
posiciones.append(i)
return posiciones

# Uso del modulo en otro archivo
import secuencias

```

```
secuenciaadn = "ATGCGTACGTTAGCTAGC"
motivo = "CGT"
print(f"Motivo encontrado en posiciones: -secuencias.buscarmotivo(secuenciaadn, motivo) ")
```

Listing 58: Análisis de datos genéticos con módulos

0.29.2. Modelado de Poblaciones

Los módulos pueden utilizarse para organizar funciones relacionadas con el modelado de poblaciones, permitiendo la reutilización del código en diferentes estudios ecológicos.

```
# Ejemplo de modulo para modelado de poblaciones: poblacion.py
def crecimientopoblacional(poblacioninicial, tasacrecimiento, tiempo):
poblacion = poblacioninicial
for   in range(tiempo):
poblacion *= (1 + tasacrecimiento)
return poblacion

# Uso del modulo en otro archivo
import poblacion

poblacioninicial = 100
tasacrecimiento = 0.05
tiempo = 10  # anos
print(f"Poblacion despues de -tiempo  anos: -poblacion.crecimientopoblacional(poblacioninicial
  , tasacrecimiento, tiempo) ")
```

Listing 59: Modelado de poblaciones con módulos

0.29.3. Análisis de Datos Ambientales

En ecología, los paquetes pueden agrupar módulos relacionados con el análisis de datos ambientales, facilitando la organización y el uso de herramientas de análisis complejas.

```
# Estructura del paquete datosambientales
# datosambientales/
# init.py
# lectura.py
# analisis.py

# datosambientales/lectura.py
import csv

def leerdatosarchivo(archivocsv):
with open(archivocsv, 'r') as archivo:
lector = csv.reader(archivo)
cabeceras = next(lector)
datos = [fila for fila in lector]
return datos

# datosambientales/analisis.py
def calcularpromedio(lecturas, columna):
valores = [float(fila[columna]) for fila in lecturas]
return sum(valores) / len(valores)

# Uso del paquete en otro archivo
from datosambientales import lectura, analisis

datos = lectura.leerdatosarchivo('datossensores.csv')
promediotemperatura = analisis.calcularpromedio(datos, 1)
print(f"Promedio de temperatura: -promediotemperatura ")
```

Listing 60: Análisis de datos ambientales con paquetes

0.29.4. Simulación de Ecosistemas

Los paquetes pueden organizar funciones y clases relacionadas con la simulación de interacciones en un ecosistema, facilitando la gestión de simulaciones complejas.

```
# Estructura del paquete ecosistema
# ecosistema/
#  init.py
#  especies.py
#  simulacion.py

# ecosistema/especies.py
class Especie:
def  init (self, nombre, cantidad):
self.nombre = nombre
self.cantidad = cantidad

# ecosistema/simulacion.py
def interaccionesecosistema(especies, interacciones, tiempo):
for   in range(tiempo):
for especie, data in especies.items():
if especie in interacciones:
for interaccion, efecto in interacciones[especie].items():
data.cantidad += data.cantidad * efecto * especies[interaccion].cantidad
return especies

# Uso del paquete en otro archivo
from ecosistema.especies import Especie
from ecosistema.simulacion import interaccionesecosistema

depredador = Especie(”Depredador”, 10)
presa = Especie(”Presa”, 50)
especies = -”depredador”: depredador, ”presa”: presa

interacciones = -
  ”depredador”: -”presa”: -0.01,
  ”presa”: -”depredador”: 0.02


resultado = interaccionesecosistema(especies, interacciones, 12)
print(f”Simulaci0n de interacciones en el ecosistema: -resultado ”)
```

Listing 61: Simulación de ecosistemas con paquetes

Programación orientada a objetos

0.30. Introducción

La programación orientada a objetos (POO) en Python permite a los desarrolladores estructurar su código utilizando clases y objetos, facilitando la creación de sistemas complejos y altamente organizados. La POO es particularmente útil en aplicaciones científicas como la biología y la ecología, donde es necesario modelar entidades y sus interacciones de manera natural y eficiente.

0.31. Importancia de la Programación Orientada a Objetos

0.31.1. Modularidad

La POO permite dividir un programa en módulos o clases independientes, lo que facilita la gestión y el desarrollo colaborativo de proyectos grandes.

0.31.2. Reutilización

Las clases y objetos pueden reutilizarse en diferentes partes de un programa o en múltiples proyectos, reduciendo la duplicación de código y mejorando la eficiencia del desarrollo.

0.31.3. Mantenibilidad

La POO mejora la mantenibilidad del código, ya que las clases encapsulan datos y métodos, permitiendo que los cambios se realicen de manera controlada y sin afectar otras partes del programa.

0.31.4. Abstracción y Encapsulamiento

La POO facilita la abstracción y el encapsulamiento, ocultando los detalles internos de las clases y exponiendo solo lo necesario a través de interfaces claras y bien definidas.

0.32. Ejemplos Prácticos en Biología y Ecología

0.32.1. Modelado de Especies

En ecología, las clases pueden utilizarse para modelar diferentes especies, encapsulando sus propiedades y comportamientos.

```
# Ejemplo de clase para modelar especies
class Especie:
def  init (self, nombre, poblacion, tasacrecimiento):
self.nombre = nombre
self.poblacion = poblacion
self.tasacrecimiento = tasacrecimiento

def crecer(self):
self.poblacion *= (1 + self.tasacrecimiento)

# Uso de la clase Especie
```

```
puma = Especie("Puma concolor", 100, 0.05)
puma.crecer()
print(f"-puma.nombre : Poblacion despues de crecer = -puma.poblacion ")
```

Listing 62: Modelado de especies

0.32.2. Simulación de Interacciones Ecológicas

Las clases pueden utilizarse para modelar interacciones complejas entre diferentes especies en un ecosistema.

```
# Ejemplo de clases para modelar interacciones ecologicas
class Especie:
def  init (self, nombre, poblacion):
self.nombre = nombre
self.poblacion = poblacion

class Depredador(Especie):
def  init (self, nombre, poblacion, tasapredacion):
super(). init (nombre, poblacion)
self.tasapredacion = tasapredacion

def depredar(self, presa):
depredacion = self.poblacion * self.tasapredacion
presa.poblacion -= depredacion
if presa.poblacion ¡ 0:
presa.poblacion = 0

class Presa(Especie):
def  init (self, nombre, poblacion, tasareproduccion):
super(). init (nombre, poblacion)
self.tasareproduccion = tasareproduccion

def reproducir(self):
self.poblacion *= (1 + self.tasareproduccion)

# Uso de las clases Depredador y Presa
lobo = Depredador("Canis lupus", 10, 0.1)
ciervo = Presa("Cervus elaphus", 50, 0.2)

# Simulacion de interacciones
for mes in range(12):
lobo.depredar(ciervo)
ciervo.reproducir()
print(f"Mes -mes + 1 : Lobos = -lobo.poblacion , Ciervos = -ciervo.poblacion ")
```

Listing 63: Simulación de interacciones ecológicas

0.32.3. Gestión de Datos Genéticos

Las clases pueden utilizarse para encapsular datos genéticos y proporcionar métodos para su análisis.

```
# Ejemplo de clase para gestionar datos geneticos
class SecuenciaGenetica:
def  init (self, secuencia):
self.secuencia = secuencia

def contarbases(self):
conteo = -"A": 0, "T": 0, "G": 0, "C": 0
for base in self.secuencia:
if base in conteo:
conteo[base] += 1
return conteo

# Uso de la clase SecuenciaGenetica
secuencia = SecuenciaGenetica("ATGCGTACGTTAGCTAGC")
conteobases = secuencia.contarbases()
print(f"Conteo de bases: -conteobases ")
```

Listing 64: Gestión de datos genéticos

0.32.4. Modelado de Ecosistemas Complejos

Las clases pueden utilizarse para modelar ecosistemas completos, encapsulando las propiedades y comportamientos de múltiples entidades y sus interacciones.

```
# Ejemplo de clases para modelar un ecosistema complejo
class Especie:
def  init (self, nombre, poblacion):
self.nombre = nombre
self.poblacion = poblacion

class Ecosistema:
def  init (self):
self.especies = []

def agregarespecie(self, especie):
self.especies.append(especie)

def simular(self, meses):
for mes in range(meses):
for especie in self.especies:
if isinstance(especie, Depredador):
for presa in self.especies:
if isinstance(presa, Presa):
especie.depredar(presa)
elif isinstance(especie, Presa):
especie.reproducir()
self.imprimirestado(mes + 1)

def imprimirestado(self, mes):
print(f"Mes -mes : ")
for especie in self.especies:
print(f"-especie.nombre :  -especie.poblacion ")

class Depredador(Especie):
def  init (self, nombre, poblacion, tasapredacion):
super(). init (nombre, poblacion)
self.tasapredacion = tasapredacion

def depredar(self, presa):
depredacion = self.poblacion * self.tasapredacion
presa.poblacion -= depredacion
if presa.poblacion ¡ 0:
presa.poblacion = 0

class Presa(Especie):
def  init (self, nombre, poblacion, tasareproduccion):
super(). init (nombre, poblacion)
self.tasareproduccion = tasareproduccion

def reproducir(self):
self.poblacion *= (1 + self.tasareproduccion)

# Uso de las clases para modelar un ecosistema complejo
lobo = Depredador("Canis lupus", 10, 0.1)
ciervo = Presa("Cervus elaphus", 50, 0.2)
ecosistema = Ecosistema()
ecosistema.agregarespecie(lobo)
ecosistema.agregarespecie(ciervo)

ecosistema.simular(12)
```

Listing 65: Modelado de ecosistemas complejos

Gestión de memoria y recolección de basura

0.33. Introducción

En Python, la gestión de memoria y la recolección de basura son mecanismos fundamentales que permiten la asignación y liberación eficiente de memoria durante la ejecución de programas. La gestión adecuada de estos recursos es crucial para asegurar el rendimiento y la estabilidad del software, especialmente en aplicaciones científicas como la biología y la ecología, donde el manejo de grandes volúmenes de datos es común.

0.34. Importancia de la Gestión de Memoria y Recolección de Basura

0.34.1. Eficiencia de Recursos

La gestión de memoria eficiente asegura que los recursos del sistema se utilicen de manera óptima, evitando el desperdicio de memoria y mejorando el rendimiento del programa.

0.34.2. Prevención de Fugas de Memoria

La recolección de basura en Python ayuda a prevenir fugas de memoria al liberar automáticamente la memoria que ya no es utilizada por el programa, asegurando que los recursos se reciclen adecuadamente.

0.34.3. Estabilidad del Programa

La administración adecuada de la memoria contribuye a la estabilidad del programa, evitando errores relacionados con la falta de memoria y asegurando un funcionamiento continuo y fiable.

0.34.4. Simplificación del Código

La recolección de basura automática simplifica el desarrollo del código, ya que los programadores no necesitan gestionar manualmente la liberación de memoria, permitiendo centrarse en la lógica del problema.

0.35. Ejemplos Prácticos en Biología y Ecología

0.35.1. Análisis de Datos Genéticos a Gran Escala

En biología molecular, el análisis de grandes volúmenes de datos genéticos requiere una gestión eficiente de la memoria para asegurar que los recursos se utilicen de manera óptima.

```
import numpy as np

# Generar un gran conjunto de datos geneticos
datosgeneticos = np.random.randint(2, size=(1000000, 100))

# Analisis: contar la frecuencia de cada base
frecuencias = np.sum(datosgeneticos, axis=0)

```

```
# Procesar los datos sin preocuparse por la gestion manual de memoria
print(f"Frecuencias de bases: -frecuencias ")
```

Listing 66: Análisis de datos genéticos a gran escala

0.35.2. Simulación de Crecimiento Poblacional en Ecosistemas

Las simulaciones a largo plazo de crecimiento poblacional requieren la liberación eficiente de memoria para asegurar que los datos históricos no ocupen memoria innecesaria.

```
class Especie:
def  init (self, nombre, poblacion):
self.nombre = nombre
self.poblacion = poblacion

def crecer(self, tasacrecimiento):
self.poblacion *= (1 + tasacrecimiento)

# Crear instancias de especies
especie = Especie("Puma concolor", 100)

# Simulacion de crecimiento
for  in range(100):
especie.crecer(0.05)
# La recoleccion de basura se encarga de la memoria utilizada por cada ciclo
print(f"Poblacion final: -especie.poblacion ")
```

Listing 67: Simulación de crecimiento poblacional en ecosistemas

0.35.3. Procesamiento de Datos Ambientales en Tiempo Real

En ecología, el procesamiento de datos ambientales en tiempo real requiere una gestión eficiente de la memoria para asegurar que los datos más recientes se analicen rápidamente sin acumular datos innecesarios.

```
import random

class Sensor:
def  init (self, id):
self.id = id
self.lecturas = []

def tomarlectura(self):
# Simular una lectura de sensor
lectura = random.uniform(20.0, 30.0)
self.lecturas.append(lectura)
# Limitar el tamano de la lista de lecturas para ahorrar memoria
if len(self.lecturas) ¿ 100:
self.lecturas.pop(0)

# Crear un sensor y tomar lecturas
sensor = Sensor(1)
for  in range(1000):
sensor.tomarlectura()
print(f"Ultima lectura: -sensor.lecturas[-1]")
```

Listing 68: Procesamiento de datos ambientales en tiempo real

0.35.4. Modelado de Redes Tróficas en Ecosistemas

El modelado de redes tróficas en ecosistemas requiere la creación y destrucción de numerosos objetos para representar las interacciones entre especies, lo que hace crucial una recolección de basura eficiente.

```
class Especie:
def  init (self, nombre):
self.nombre = nombre
self.depredadores = []
```

```
    self.presas = []

    def agregardepredador(self, depredador):
    self.depredadores.append(depredador)

    def agregarpresa(self, presa):
    self.presas.append(presa)

    # Crear instancias de especies y sus interacciones
    lobo = Especie("Lobo")
    ciervo = Especie("Ciervo")
    conejo = Especie("Conejo")

    lobo.agregarpresa(ciervo)
    ciervo.agregarpresa(conejo)
    ciervo.agregardepredador(lobo)
    conejo.agregardepredador(ciervo)

    # Eliminar objetos y liberar memoria automaticamente
    del conejo
    print(f"Depredadores del ciervo: -[dep.nombre for dep in ciervo.depredadores] ")
```

Listing 69: Modelado de redes tróficas en ecosistemas

Librerías estándar de Python

0.36. Introducción

Las librerías estándar de Python son colecciones de módulos que se incluyen con la distribución estándar de Python. Estas librerías ofrecen soluciones para tareas comunes y especializadas, permitiendo a los desarrolladores escribir código más limpio y eficiente sin necesidad de reinventar la rueda. En biología y ecología, estas librerías pueden simplificar significativamente el procesamiento de datos, el análisis y la modelización.

0.37. Importancia de las Librerías Estándar

0.37.1. Productividad Mejorada

Las librerías estándar permiten a los desarrolladores trabajar de manera más eficiente, proporcionando herramientas listas para usar que reducen la cantidad de código que se debe escribir desde cero.

0.37.2. Confiabilidad y Mantenimiento

Al usar librerías estándar, los desarrolladores pueden confiar en el código probado y mantenido por la comunidad de Python, lo que reduce la probabilidad de errores y mejora la mantenibilidad del software.

0.37.3. Compatibilidad y Portabilidad

Las librerías estándar están diseñadas para ser compatibles con múltiples plataformas, lo que garantiza que el código desarrollado sea portable y pueda ejecutarse en diferentes entornos sin modificaciones significativas.

0.37.4. Amplia Gama de Funcionalidades

Las librerías estándar cubren una amplia gama de funcionalidades, desde manipulación de archivos y procesamiento de texto hasta cálculos matemáticos y operaciones de red, proporcionando herramientas para casi cualquier tarea.

0.38. Ejemplos Prácticos en Biología y Ecología

0.38.1. Manipulación de Archivos CSV con `csv`

En biología y ecología, es común trabajar con datos almacenados en archivos CSV. La librería `csv` facilita la lectura y escritura de estos archivos.

```
import csv

# Lectura de un archivo CSV
with open('datosgeneticos.csv', newline='') as archivo:
lector = csv.reader(archivo)
for fila in lector:
print(fila)

# Escritura en un archivo CSV
```

```
with open('resultados.csv', mode='w', newline='') as archivo:
escritor = csv.writer(archivo)
escritor.writerow(['Gen', 'Mutacion'])
escritor.writerow(['BRCA1', 'Positiva'])
```

Listing 70: Manipulación de archivos CSV

0.38.2. Procesamiento de Datos Numéricos con `math`

La librería `math` proporciona funciones matemáticas básicas y avanzadas, esenciales para el análisis cuantitativo en biología y ecología.

```
import math

# Calcular la tasa de crecimiento poblacional
poblacioninicial = 100
tasacrecimiento = 0.05
tiempo = 10  # anos

poblacionfinal = poblacioninicial * math.exp(tasacrecimiento * tiempo)
print(f"Poblacion despues de -tiempo  anos: -poblacionfinal ")
```

Listing 71: Procesamiento de datos numéricos

0.38.3. Análisis de Fechas y Tiempos con `datetime`

El módulo `datetime` es útil para manejar fechas y tiempos, una tarea común en estudios de campo y monitoreo ambiental.

```
from datetime import datetime, timedelta

# Calcular la fecha y hora actual
ahora = datetime.now()
print(f"Fecha y hora actual: -ahora ")

# Calcular una fecha futura basada en un intervalo de tiempo
diasfuturos = 30
fechafutura = ahora + timedelta(days=diasfuturos)
print(f"Fecha dentro de -diasfuturos  dias: -fechafutura ")
```

Listing 72: Análisis de fechas y tiempos

0.38.4. Serialización de Datos con `json`

El módulo `json` permite la serialización y deserialización de datos en formato JSON, facilitando el intercambio de datos entre diferentes sistemas y plataformas.

```
import json

# Datos de observacion de especies en formato diccionario
datosobservacion = -
  "especie": "Puma concolor",
  "ubicacion": "Parque Nacional",
  "cantidad": 2


# Serializar los datos a una cadena JSON
jsondata = json.dumps(datosobservacion)
print(f"Datos en formato JSON: -jsondata ")

# Deserializar la cadena JSON a un diccionario
datosobservacionrecuperados = json.loads(jsondata)
print(f"Datos deserializados: -datosobservacionrecuperados ")
```

Listing 73: Serialización de datos con JSON

Taller1-Cap1

2.1 Obtener el Área de un Triángulo

Pseudocódigo

```
Inicio
Leer base, altura
Área = (base * altura) / 2
Mostrar Área
Fin
```

2.2 Convertir Dinero Mexicano a Dólares

Pseudocódigo

```
Inicio
Leer cantidad en pesos
Dólares = Pesos / Tipo de cambio
Mostrar Dólares
Fin
```

Pseudocódigo

```
Inicio
Leer ano de nacimiento
Edad = Ano actual - Ano de nacimiento
Mostrar Edad
Fin
```

2.4 Determinar el Cobro de un Estacionamiento

Pseudocódigo

```
Inicio
Leer horas de uso
Cobro = Tarifa * Horas
Mostrar Cobro
Fin
```

2.5 Determinar el Costo de Trabajos de Pintura

Pseudocódigo

```
Inicio
```

```
Leer metros cuadrados
Costo = Precio por m2 * Metros cuadrados
Mostrar Costo
Fin
```

Taller2-Cap2

```
edad = int(input("Ingrese la edad de la planta en dias: "))
if edad ¡ 30:
print("Plantula")
elif edad ¡ 180:
print("Joven")
else:
print("Adulta")
```

Listing 74: Biología: Clasificación de Edad de una Planta

```
temp = float(input("Ingrese la temperatura del agua en Celsius: "))
if temp ¡ 5:
print("Agua fria")
elif temp ¡= 25:
print("Agua templada")
else:
print("Agua caliente")
```

Listing 75: Ecología: Estado de un Río según la Temperatura del Agua

```
concentracion = float(input("Ingrese la concentracion de la solucion en porcentaje: "))
if concentracion ¡ 0.5:
print("Solucion diluida")
elif concentracion ¡= 10:
print("Solucion moderada")
else:
print("Solucion concentrada")
```

Listing 76: Química Farmacéutica: Determinación de la Concentración de una Solución

```
peso = float(input("Ingrese el peso del animal en kg: "))
if peso ¡ 1:
print("Pequeno")
elif peso ¡= 20:
print("Mediano")
else:
print("Grande")
```

Listing 77: Ecología: Clasificación de un Animal según su Peso

```
edad = int(input("Ingrese la edad de la planta en dias: "))
if edad ¡ 30:
print("Plantula")
elif edad ¡ 180:
print("Joven")
else:
print("Adulta")
```

Listing 78: Biología: Clasificación de Edad de una Planta

```
temp = float(input("Ingrese la temperatura del agua en Celsius: "))
if temp ¡ 5:
print("Agua fria")
elif temp ¡= 25:
```

```
print(”Agua templada”)
else:
print(”Agua caliente”)
```

Listing 79: Ecología: Estado de un Río según la Temperatura del Agua

```
concentracion = float(input(”Ingrese la concentracion de la solucion en porcentaje: ”))
if concentracion ¡ 0.5:
print(”Solucion diluida”)
elif concentracion ¡= 10:
print(”Solucion moderada”)
else:
print(”Solucion concentrada”)
```

Listing 80: Química Farmacéutica: Determinación de la Concentración de una Solución

```
peso = float(input(”Ingrese el peso del animal en kg: ”))
if peso ¡ 1:
print(”Pequeno”)
elif peso ¡= 20:
print(”Mediano”)
else:
print(”Grande”)
```

Listing 81: Ecología: Clasificación de un Animal según su Peso

```
ph = float(input(”Ingrese el pH del suelo: ”))
if ph ¡ 5.5:
print(”Suelo acido”)
elif ph ¡= 7.5:
print(”Suelo neutro”)
else:
print(”Suelo alcalino”)
```

Listing 82: Biología: Evaluación del pH del Suelo

```
velocidad = float(input(”Ingrese la velocidad del viento en km/h: ”))
if velocidad ¡ 20:
print(”Viento debil”)
elif velocidad ¡= 40:
print(”Viento moderado”)
else:
print(”Viento fuerte”)
```

Listing 83: Ecología: Clasificación de la Velocidad del Viento

```
pureza = float(input(”Ingrese la pureza del compuesto en porcentaje: ”))
if pureza ¡ 90:
print(”Impureza alta”)
elif pureza ¡= 99:
print(”Pureza moderada”)
else:
print(”Pureza alta”)
```

Listing 84: Química Farmacéutica: Evaluación de la Pureza de un Compuesto

Taller1-Cap3

2.1 Obtener el Área de un Triángulo

Pseudocódigo

```
Inicio
Leer base, altura
Área = (base * altura) / 2
Mostrar Área
Fin
```

2.2 Convertir Dinero Mexicano a Dólares

Pseudocódigo

```
Inicio
Leer cantidad en pesos
Dólares = Pesos / Tipo de cambio
Mostrar Dólares
Fin
```

Pseudocódigo

```
Inicio
Leer ano de nacimiento
Edad = Ano actual - Ano de nacimiento
Mostrar Edad
Fin
```

2.4 Determinar el Cobro de un Estacionamiento

Pseudocódigo

```
Inicio
Leer horas de uso
Cobro = Tarifa * Horas
Mostrar Cobro
Fin
```

2.5 Determinar el Costo de Trabajos de Pintura

Pseudocódigo

```
Inicio
```

```
Leer metros cuadrados
Costo = Precio por m2 * Metros cuadrados
Mostrar Costo
Fin
```

Taller1-Cap4

2.1 Obtener el Área de un Triángulo

Pseudocódigo

```
Inicio
Leer base, altura
Área = (base * altura) / 2
Mostrar Área
Fin
```

2.2 Convertir Dinero Mexicano a Dólares

Pseudocódigo

```
Inicio
Leer cantidad en pesos
Dólares = Pesos / Tipo de cambio
Mostrar Dólares
Fin
```

Pseudocódigo

```
Inicio
Leer ano de nacimiento
Edad = Ano actual - Ano de nacimiento
Mostrar Edad
Fin
```

2.4 Determinar el Cobro de un Estacionamiento

Pseudocódigo

```
Inicio
Leer horas de uso
Cobro = Tarifa * Horas
Mostrar Cobro
Fin
```

2.5 Determinar el Costo de Trabajos de Pintura

Pseudocódigo

```
Inicio
```

```
Leer metros cuadrados
Costo = Precio por m2 * Metros cuadrados
Mostrar Costo
Fin
```

Taller1-Cap5

2.1 Obtener el Área de un Triángulo

Pseudocódigo

```
Inicio
Leer base, altura
Área = (base * altura) / 2
Mostrar Área
Fin
```

2.2 Convertir Dinero Mexicano a Dólares

Pseudocódigo

```
Inicio
Leer cantidad en pesos
Dólares = Pesos / Tipo de cambio
Mostrar Dólares
Fin
```

Pseudocódigo

```
Inicio
Leer ano de nacimiento
Edad = Ano actual - Ano de nacimiento
Mostrar Edad
Fin
```

2.4 Determinar el Cobro de un Estacionamiento

Pseudocódigo

```
Inicio
Leer horas de uso
Cobro = Tarifa * Horas
Mostrar Cobro
Fin
```

2.5 Determinar el Costo de Trabajos de Pintura

Pseudocódigo

```
Inicio
```

```
Leer metros cuadrados
Costo = Precio por m2 * Metros cuadrados
Mostrar Costo
Fin
```

Taller1-Cap6

2.1 Obtener el Área de un Triángulo

Pseudocódigo

```
Inicio
Leer base, altura
Área = (base * altura) / 2
Mostrar Área
Fin
```

2.2 Convertir Dinero Mexicano a Dólares

Pseudocódigo

```
Inicio
Leer cantidad en pesos
Dólares = Pesos / Tipo de cambio
Mostrar Dólares
Fin
```

Pseudocódigo

```
Inicio
Leer ano de nacimiento
Edad = Ano actual - Ano de nacimiento
Mostrar Edad
Fin
```

2.4 Determinar el Cobro de un Estacionamiento

Pseudocódigo

```
Inicio
Leer horas de uso
Cobro = Tarifa * Horas
Mostrar Cobro
Fin
```

2.5 Determinar el Costo de Trabajos de Pintura

Pseudocódigo

```
Inicio
```

```
Leer metros cuadrados
Costo = Precio por m2 * Metros cuadrados
Mostrar Costo
Fin
```

Taller1-Cap7

2.1 Obtener el Área de un Triángulo

Pseudocódigo

```
Inicio
Leer base, altura
Área = (base * altura) / 2
Mostrar Área
Fin
```

2.2 Convertir Dinero Mexicano a Dólares

Pseudocódigo

```
Inicio
Leer cantidad en pesos
Dólares = Pesos / Tipo de cambio
Mostrar Dólares
Fin
```

Pseudocódigo

```
Inicio
Leer ano de nacimiento
Edad = Ano actual - Ano de nacimiento
Mostrar Edad
Fin
```

2.4 Determinar el Cobro de un Estacionamiento

Pseudocódigo

```
Inicio
Leer horas de uso
Cobro = Tarifa * Horas
Mostrar Cobro
Fin
```

2.5 Determinar el Costo de Trabajos de Pintura

Pseudocódigo

```
Inicio
```

```
Leer metros cuadrados
Costo = Precio por m2 * Metros cuadrados
Mostrar Costo
Fin
```

Taller1-Cap8

2.1 Obtener el Área de un Triángulo

Pseudocódigo

```
Inicio
Leer base, altura
Área = (base * altura) / 2
Mostrar Área
Fin
```

2.2 Convertir Dinero Mexicano a Dólares

Pseudocódigo

```
Inicio
Leer cantidad en pesos
Dólares = Pesos / Tipo de cambio
Mostrar Dólares
Fin
```

Pseudocódigo

```
Inicio
Leer ano de nacimiento
Edad = Ano actual - Ano de nacimiento
Mostrar Edad
Fin
```

2.4 Determinar el Cobro de un Estacionamiento

Pseudocódigo

```
Inicio
Leer horas de uso
Cobro = Tarifa * Horas
Mostrar Cobro
Fin
```

2.5 Determinar el Costo de Trabajos de Pintura

Pseudocódigo

```
Inicio
```

```
Leer metros cuadrados
Costo = Precio por m2 * Metros cuadrados
Mostrar Costo
Fin
```

Taller1-Cap9

2.1 Obtener el Área de un Triángulo

Pseudocódigo

```
Inicio
Leer base, altura
Área = (base * altura) / 2
Mostrar Área
Fin
```

2.2 Convertir Dinero Mexicano a Dólares

Pseudocódigo

```
Inicio
Leer cantidad en pesos
Dólares = Pesos / Tipo de cambio
Mostrar Dólares
Fin
```

Pseudocódigo

```
Inicio
Leer ano de nacimiento
Edad = Ano actual - Ano de nacimiento
Mostrar Edad
Fin
```

2.4 Determinar el Cobro de un Estacionamiento

Pseudocódigo

```
Inicio
Leer horas de uso
Cobro = Tarifa * Horas
Mostrar Cobro
Fin
```

2.5 Determinar el Costo de Trabajos de Pintura

Pseudocódigo

```
Inicio
```

```
Leer metros cuadrados
Costo = Precio por m2 * Metros cuadrados
Mostrar Costo
Fin
```

Taller1-Cap10

2.1 Obtener el Área de un Triángulo

Pseudocódigo

```
Inicio
Leer base, altura
Área = (base * altura) / 2
Mostrar Área
Fin
```

2.2 Convertir Dinero Mexicano a Dólares

Pseudocódigo

```
Inicio
Leer cantidad en pesos
Dólares = Pesos / Tipo de cambio
Mostrar Dólares
Fin
```

Pseudocódigo

```
Inicio
Leer ano de nacimiento
Edad = Ano actual - Ano de nacimiento
Mostrar Edad
Fin
```

2.4 Determinar el Cobro de un Estacionamiento

Pseudocódigo

```
Inicio
Leer horas de uso
Cobro = Tarifa * Horas
Mostrar Cobro
Fin
```

2.5 Determinar el Costo de Trabajos de Pintura

Pseudocódigo

```
Inicio
```

```
Leer metros cuadrados
Costo = Precio por m2 * Metros cuadrados
Mostrar Costo
Fin
```

Taller1-Cap11

2.1 Obtener el Área de un Triángulo

Pseudocódigo

```
Inicio
Leer base, altura
Área = (base * altura) / 2
Mostrar Área
Fin
```

2.2 Convertir Dinero Mexicano a Dólares

Pseudocódigo

```
Inicio
Leer cantidad en pesos
Dólares = Pesos / Tipo de cambio
Mostrar Dólares
Fin
```

Pseudocódigo

```
Inicio
Leer ano de nacimiento
Edad = Ano actual - Ano de nacimiento
Mostrar Edad
Fin
```

2.4 Determinar el Cobro de un Estacionamiento

Pseudocódigo

```
Inicio
Leer horas de uso
Cobro = Tarifa * Horas
Mostrar Cobro
Fin
```

2.5 Determinar el Costo de Trabajos de Pintura

Pseudocódigo

```
Inicio
```

```
Leer metros cuadrados
Costo = Precio por m2 * Metros cuadrados
Mostrar Costo
Fin
```

Taller1-Cap12

2.1 Obtener el Área de un Triángulo

Pseudocódigo

```
Inicio
Leer base, altura
Área = (base * altura) / 2
Mostrar Área
Fin
```

2.2 Convertir Dinero Mexicano a Dólares

Pseudocódigo

```
Inicio
Leer cantidad en pesos
Dólares = Pesos / Tipo de cambio
Mostrar Dólares
Fin
```

Pseudocódigo

```
Inicio
Leer ano de nacimiento
Edad = Ano actual - Ano de nacimiento
Mostrar Edad
Fin
```

2.4 Determinar el Cobro de un Estacionamiento

Pseudocódigo

```
Inicio
Leer horas de uso
Cobro = Tarifa * Horas
Mostrar Cobro
Fin
```

2.5 Determinar el Costo de Trabajos de Pintura

Pseudocódigo

```
Inicio
```

```
Leer metros cuadrados
Costo = Precio por m2 * Metros cuadrados
Mostrar Costo
Fin
```

Evaluaciones Semestre 01 — 2026

Primera evaluación de programación Bio-Python (Quiz Uno)

El siguiente enlace permite acceder al primer quiz cuyo valor es de 5 por ciento:	https://jestrada2020.github.io/PrimerQuizSemestre01BioPython2026version1/

También puedes usar el código QR para acceder al primer quiz, cuyo valor es de 5 por ciento:

Segunda evaluación de programación Bio-Python (Parcial Uno)

El siguiente enlace permite acceder al primer parcial cuyo valor es de 20 por ciento:	https://jestrada2020.github.io/PrimerParcialBioPythnoVersion12026/

También puedes usar el código QR para acceder al primer parcial, cuyo valor es de 20 por ciento:

Bibliografía

[1] James Stewart, Lothar Redlin, Saleem Watson, Héctor Vidauri, Alejandro Alfaro. PRECÁLCULO Matemáticas para el cálculo, quinta edición. Thomson/Brooks Cole © 2006 ISBN: 0-534-49277-0

[2] Zill, D. G.,& Dewar. J. M. (2012). *Álgebra, trigonometría y geometría analítica.* McGraw Hill.

[3] Swokowski, Earl W. y Jeffery A. Cole (2009) *Álgebra y Trigonometría con Geometría Analítica*, Décimo Segunda edición, Grupo Editorial Iberoamérica. México.

[4] Swokowski. E., & COLE, J. (1996). *Álgebra y trigonometría.* Grupo Editorial Iberoamérica. México.

[5] Sullivan, J. (2006).*Álgebra y Trigonometría.* Pearson educación

[6] Dominguez, F. A. Nieves, *Métodos Numéricos con Aplicaciones a la Ingeniería*, editorial Sisa.

ProgUno UCES

Enero

Lun	Mar	Mie	Jue	Vie	Sab	Dom
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31				

ProgUno UCES

Febrero

Lun	Mar	Mie	Jue	Vie	Sab	Dom
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28				

ProgUno UCES

Marzo

Lun	Mar	Mie	Jue	Vie	Sab	Dom
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

ProgUno UCES

Abril

Lun	Mar	Mie	Jue	Vie	Sab	Dom
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30					

ProgUno UCES

Mayo

Lun	Mar	Mie	Jue	Vie	Sab	Dom
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

Fechas relevantes en colores:

Parcial color azul

Quiz color azul

CLAVE PARA MATRICULA VIRTUAL:
PROGRAMACIONJULIO17