

A graphic on the left side of the slide consists of four overlapping horizontal bars in purple, orange, yellow, and blue, with an orange arrow pointing to the right. The text 'Agencia de Aprendizaje a lo largo de la vida' is written across these bars in white.

Agencia de  
Aprendizaje  
a lo largo  
de la vida

# FULL STACK PYTHON

## Clase 32

PYTHON 8

# Manejo de excepciones

## Módulos y paquetes



# Les damos la bienvenida

Vamos a comenzar a grabar la clase

**Clase 31****Herencia y Polimorfismo**

- Herencia.
- Polimorfismo.
- Herencia simple y múltiple.
- Clases Abstractas.
- Diagrama de Clases.

**Clase 32****Manejo de excepciones y Módulos y paquetes**

- Manejo de excepciones.
- Errores vs. excepciones.
- Múltiples excepciones, invocación y creación de excepciones propias.
- Módulos y packages.
- Librerías.
- Collections, datetime, math y módulo random.

**Clase 33****Proyecto Final con Python**

- Introducción al Framework a utilizar.
- Introducción al Proyecto CRUD.
- Instalación y configuración.
- Creación de un proyecto utilizando el Framework.

# Errores y excepciones

Las **excepciones** son una herramienta muy importante para determinar el comportamiento de un programa cuando se produce un **error**.

Presente en la mayoría de los lenguajes de programación modernos, Python incluido, las excepciones proveen mecanismos para “atrapar” el error y desviar el flujo del programa a una bloque de instrucciones que puedan tratar adecuadamente el problema.

Dado que es difícil escribir grandes porciones de código 100% libres de errores, o prever todas las acciones que puede realizar el usuario, conocer el uso de excepciones es fundamental para cualquier programador.

# Errores

Los **errores** detienen la ejecución del programa. Pueden originarse de varias maneras. Entre los más comunes se encuentran los **errores de sintaxis**, que se identifican con el código **SyntaxError** y tienen lugar cuando no se respetan las reglas sintácticas de Python. Por ejemplo, al “olvidar” de cerrar un paréntesis:

## Programa

```
# Programa principal:  
a = 2  
print("Hola"
```

## Terminal

```
File "error.py", line 3  
    print("Hola"  
          ^  
SyntaxError: '(' was never closed
```

*El intérprete reproduce la línea responsable del error y muestra el lugar donde se detectó el error.*

Código del error

# Errores

Cuando se intenta operar con tipos de datos incompatibles entre sí, se producen **errores de tipo (TypeError)**:

## Programa

```
# Programa principal:  
print(2 + "2")
```

## Terminal

```
File "error.py", line 1, in <module>  
    print(2 + "2")  
TypeError: unsupported operand type(s)  
for +: 'int' and 'str'
```

*Python es muy preciso a la hora de informar del error producido. No puede lidiar con tipos diferentes (enteros y cadenas).*

Si se trata de dividir por cero, el error es del tipo **ZeroDivisionError**:

## Programa

```
# Programa principal:  
a = 4; b = 0  
print(a/b)
```

## Terminal

```
File "error.py", line 3, in <module>  
    print(a/b)  
ZeroDivisionError: division by zero
```

*Cuando el usuario es el que ingresa los datos, puede ocurrir que tenga lugar un error como este.*

# Errores

Los **errores semánticos** suelen ser difíciles de detectar. No son errores de escritura (sintácticos) sino “de lógica”. Por ejemplo, cuando intentamos eliminar un elemento de una lista vacía:

## Programa

```
mi_lista = []  
mi_lista.pop()
```

## Terminal

```
File "error.py", line 2, in <module>  
    mi_lista.pop()  
IndexError: pop from empty list
```

Al referenciar un método o función que no existe obtenemos un **NameError**:

## Programa

```
print("¡ups!")
```

## Terminal

```
File "error.py", line 1, in <module>  
    print("¡ups!")  
NameError: name 'print' is not defined. Did you mean: 'print'?
```



# Errores

Al ingresar un valor con la función `input()`, este siempre es del tipo `string`. Si se intenta operar entre este valor y otros números sin realizar la conversión de tipos, tendremos también un fallo **TypeError**, que no será detectado por los editores de código:

## Programa

```
n = input("Ingrese un número: ")
mitad = n / 2
print(f"{n}/2 = {mitad}")
```

## Terminal

```
File "error.py", line 2, in <module>
    mitad = n / 2
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Existen docenas de errores más, y para cada uno de ellos Python provee un código, formalmente llamado **Exception**, que permite determinar que clase de error se ha cometido y, lo más importante, hacer algo para solucionarlo.

# Excepciones | Bloques try - except

Las excepciones son bloques de código que permiten continuar con la ejecución de un programa pese a que ocurra un error.

Se coloca el código que puede fallar dentro de un **bloque try** y a continuación un **bloque except** con el código necesario para tratar la situación excepcional provocada por el fallo.

## Bloques try - except

```
try:
    n = input("Ingrese un número: ")
    m = 4
    print(f"{n}/{m} = {n/m}")
except:
    print("Ha ocurrido un error.")
    print("Introduzca un NÚMERO.")
```

## Terminal

```
Ingrese un número: hola
Ha ocurrido un error.
Introduzca un NÚMERO.
```

# Excepciones | Bloques try - except

Las excepciones se pueden utilizar dentro de un bucle **while**, repitiendo la lectura por teclado hasta que se ingrese un número y entonces romper el bucle con un **break**:

## Programa

```
while True:
    try:
        n = float(input("Ingrese un número: "))
        m = 4
        print(f"{n}/{m} = {n/m}")
        break # Sale del bucle
    except:
        print("Ha ocurrido un error.")
        print("Introduzca un NÚMERO")
```

## Terminal

```
Ingrese un número: árbol
Ha ocurrido un error.
Introduzca un NÚMERO
Ingrese un número: tres
Ha ocurrido un error.
Introduzca un NÚMERO
Ingrese un número: 3
3.0/4 = 0.75
```

# Excepciones | Bloques try - except - else

El bloque opcional **else** permite incluir código que se ejecuta en caso de que la excepción no se produzca:

## Programa

```
while True:
    try:
        n = float(input("Ingrese un número: "))
        m = 4
        print(f"{n}/{m} = {n/m}")
    except:
        print("Ha ocurrido un error.")
        print("Introduzca un NÚMERO")
    else:
        print("Todo funcionó correctamente")
        break
```

## Terminal

```
Ingrese un número: uno
Ha ocurrido un error.
Introduzca un NÚMERO
Ingrese un número: 32
32.0/4 = 8.0
Todo funcionó correctamente
```

# Excepciones | Bloques try - except - finally

Finalmente, puede ubicarse un bloque **finally** donde se escriben las sentencias de finalización, que son típicamente acciones de limpieza. La particularidad del bloque **finally** es que se ejecuta siempre, haya surgido una excepción o no.

Si hay un bloque **except**, no es necesario que esté presente el **finally**, y es posible tener un bloque **try** sólo con **finally**, sin **except**.

Con este grupo de opciones, **try** se convierte en un excelente aliado para manejar los errores que puedan ocurrir durante la ejecución del programa.

# Excepciones | Bloques try - except - finally

## Programa

```
while True:
    try:
        n = float(input("Ingrese un número: "))
        m = 4
        print(f"{n}/{m} = {n/m}")
    except:
        print("Ha ocurrido un error.")
        print("Introduzca un NÚMERO")
    else:
        print(f"Se ingresó el número {n}")
        break
    finally:
        print("--Finalizado el intento")
```

## Terminal

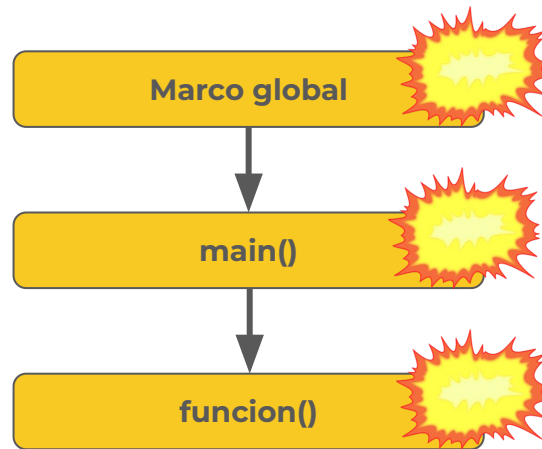
```
Ingrese un número: hola
Ha ocurrido un error.
Introduzca un NÚMERO
--Finalizado el intento
```

```
Ingrese un número: otro
Ha ocurrido un error.
Introduzca un NÚMERO
--Finalizado el intento
```

```
Ingrese un número: 120
120.0/4 = 30.0
Se ingresó el número 120.0
--Finalizado el intento
```

# Propagación de excepciones

Si dentro de una función surge una excepción y la función no posee código para manejarla, esta se **propaga** hacia la función que la invocó. Si esta otra tampoco puede hacerlo, la excepción continúa propagándose hasta llegar a la función inicial. Si esta tampoco la maneja se interrumpe la ejecución del programa.



# Propagación de excepciones

## Programa

```
def funcion(x, y):  
    print("Antes")  
    div = x/y  
    print("Después")  
    return div  
  
def main():  
    x = float(input("x: "))  
    y = float(input("y: "))  
    print(funcion(x,y))  
    print("Finalizado")  
  
main()
```

Si se introduce 0 como divisor se produce un error **ZeroDivisionError** que se propaga a las demás funciones. La terminal muestra el “recorrido” del error:

## Terminal

```
Traceback (most recent call last):  
  File "prueba.py", line 13, in <module>  
    main()  
  File "prueba.py", line 10, in main  
    print(funcion(x,y))  
  File "prueba.py", line 3, in funcion  
    div = x/y  
ZeroDivisionError: float division by zero
```



# Invocación de excepciones con raise

Utilizando **raise** se puede lanzar una excepción pasándole el identificador. Y se puede añadir un except para tratar esta excepción que se ha lanzado:

## Programa principal

```
def mi_funcion(algo=None):  
    try:  
        if algo is None:  
            raise ValueError("Error! No se permite un valor nulo")  
    except ValueError:  
        print("Error! No se permite un valor nulo (desde la excepción)")  
  
mi_funcion()
```

## Terminal

```
Error! No se permite un valor nulo (desde la excepción)
```

# Excepciones múltiples

Dado que dentro de un mismo bloque **try** pueden producirse excepciones de distinto tipo, es posible utilizar varios bloques **except**, cada uno para capturar un tipo distinto de excepción.

Esto se hace especificando a continuación de la sentencia **except** el nombre de la excepción que se pretende capturar. Un mismo bloque **except** puede atrapar varios tipos de excepciones, lo cual se hace especificando los nombres de las excepciones separados por comas a continuación de la palabra **except**. Es importante destacar que si bien luego de un bloque **try** puede haber varios bloques **except**, se ejecutará, a lo sumo, uno de ellos.

# Excepciones múltiples | Tipos de excepciones

Estas son las excepciones que se pueden producir:

ArithmeticError  
AssertionError  
AttributeError  
BaseException  
BufferError  
ChildProcessError  
ConnectionAbortedError  
ConnectionError  
ConnectionRefusedError  
ConnectionResetError  
DeprecationWarning  
EOFError  
EnvironmentError  
FileExistsError

FileNotFoundError  
FloatingPointError  
IOError  
ImportError  
IndentationError  
IndexError  
InterruptedError  
KeyError  
KeyboardInterrupt  
ModuleNotFoundError  
NameError  
NotImplementedError  
PermissionError  
RecursionError

ReferenceError  
RuntimeError  
StopIteration  
TypeError  
UnboundLocalError  
UnicodeDecodeError  
UnicodeEncodeError  
UnicodeError  
UnicodeTranslateError  
UnicodeWarning  
UserWarning  
ValueError  
Warning  
ZeroDivisionError

# Excepciones múltiples

También es posible asignar una excepción a una variable. De esta forma se puede analizar el tipo de error que ha ocurrido gracias a su identificador:

## Programa

```
try:
    # En el input no convertimos la cadena a número!
    n = input("Ingrese un número: ")
    5/n
# Guardamos la excepción en la variable e
except Exception as e:
    print("Ha ocurrido un error =>", type(e).__name__)
```

## Terminal

```
Ingrese un número: 3
Ha ocurrido un error => TypeError
```

Cada error tiene un identificador único que puede verse utilizando la sintaxis:

## Programa

```
print(type(e))
```

## Terminal

```
<class 'TypeError'>
```

# Excepciones múltiples

Se pueden utilizar múltiples **except**, dejando en último lugar **Exception** que engloba cualquier tipo de error (si se coloca antes, las demás excepciones nunca se ejecutarán):

## Excepciones múltiples

```
try:
    n = float(input("Ingrese un número divisor: "))
    5/n
except TypeError:
    print("No se puede dividir el número entre una cadena")
except ValueError:
    print("Debes introducir una cadena que sea un número")
except ZeroDivisionError:
    print("No se puede dividir por cero, prueba otro número")
except Exception as e:
    print("Ha ocurrido un error no previsto", type(e).__name__)
```

# Invocación de excepciones con raise

**raise** también se utiliza para crear una excepción. Se puede definir qué tipo de error generar y el texto para mostrar al usuario. Si no se detalla el tipo de excepción, se relanza la última excepción producida. En el ejemplo se lanza una excepción del tipo **TypeError** si x no es un entero:

## Programa principal

```
x = "Hola mundo"
if not type(x) is int:
    raise TypeError("x debe ser un entero")
```

## Terminal

```
TypeError: x debe ser un entero
```

# Excepciones | Instrucción assert

Se pueden utilizar múltiples **except**, dejando en último lugar **Exception** que engloba cualquier tipo de error (si se coloca antes, las demás excepciones nunca se ejecutarán):

# Módulos y paquetes

En Python los módulos son archivos que contienen definiciones que se pueden importar en otros scripts para reutilizar sus funcionalidades.

Un módulo es un archivo de Python cuyos objetos (funciones, clases, excepciones, etc.) pueden ser accedidos desde otro script. Constituye una muy buena herramienta para organizar el código en proyectos grandes o complejos.



# Módulos

Crear un módulo en Python es tan sencillo como crear un script, sólo tenemos que añadir alguna función a un fichero con la extensión .py, por ejemplo saludos.py:

Módulo saludos.py

```
def saludar():  
    print("Hola, te estoy saludando desde la función saludar()")
```

Los módulos se puede utilizar desde otro script, siempre que esté en la misma carpeta, agregandolo con **import**:

Programa principal

```
import saludos  
saludos.saludar()
```

# Módulos

Es posible importar funciones directamente, ahorrando memoria. Para ello se utiliza la sintaxis **from ... import ...** :

Programa principal

```
from saludos import saludar  
saludar()
```

Para importar todas las funciones con la sintaxis **from import** se utiliza un asterisco:

Programa principal

```
from saludos import *  
saludar()
```

# Módulos

Además de funciones, un módulo puede contener clases:

Módulo saludos.py

```
class Saludo():  
    def __init__(self):  
        print("Hola, te estoy saludando desde el __init__")
```

Igual que con las funciones, tendremos que importar primero al módulo para utilizar la clase en nuestro programa:

Programa principal

```
from saludos import Saludo  
s = Saludo()
```

# Paquetes

Utilizar paquetes permite unificar distintos módulos bajo un mismo nombre, con una jerarquía de módulos y submódulos (o subpaquetes). Permiten distribuir y manejar el código como si fuesen librerías instalables de Python.

Para crear un paquete se guarda un archivo vacío con el nombre `__init__.py` en el directorio donde se encuentran los módulos a agrupar. Este directorio se debe encontrar dentro del directorio en el que se aloja el script principal.

Esta jerarquía de archivos y carpetas permite acceder a los módulos, pero esta vez indicando el nombre del paquete y del módulo, también con la instrucción **import**.

# Paquetes

Se accede a los módulos con **from import**:

## Programa principal

```
from paquete.saludos import Saludos  
s = Saludo()
```

```
script.py  
paquete/  
    __init__.py  
    saludos.py
```

*Estructura de carpetas necesaria para crear un paquete. El archivo `__init__.py` es el que permite a Python tratar una carpeta como un paquete.*

Esta jerarquía de carpetas se puede expandir tanto como sea necesario creando subpaquetes, siempre añadiendo el fichero `init` en cada uno de ellos.

```
script.py  
paquete/  
    __init__.py  
    adios/  
        __init__.py  
        despedidas.py  
    hola/  
        __init__.py  
        saludos.py
```

# Paquetes | Ejemplo

## Contenido de paquete/hola/saludos.py

```
def saludar():  
    print("Hola, te estoy saludando desde la función saludar() del módulo saludos")  
  
class Saludo():  
    def __init__(self):  
        print("Hola, te estoy saludando desde el __init__ de la clase Saludo")
```

## Programa principal

```
from paquete.hola.saludos import saludar  
from paquete.adios.despedidas import Despedida  
  
saludar()  
Despedida()
```

## Paquete/adios/despedidas.py

```
def despedir():  
    print("Adiós, me estoy despidiendo desde la función despedir() del módulo despedidas")  
  
class Despedida():  
    def __init__(self):  
        print("Adiós, me estoy despidiendo desde el __init__ de la clase Despedida")
```

## Terminal

```
Hola, te estoy saludando desde la función  
saludar() del módulo saludos
```

```
Adiós, me estoy despidiendo desde el __init__  
de la clase Despedida
```

# Módulos esenciales

- **copy:** Funciones para crear copias de variables referenciadas en memoria, como colecciones y objetos.
- **collections:** Cuenta con diferentes estructuras de datos.
- **datetime:** Maneja tipos de datos referidos a las fechas/horas.
- **html, xml y json:** Permiten manejar cómodamente estructuras de datos html, xml y json.
- **math:** Incluye varias funciones matemáticas.
- **random:** Permite generar contenidos aleatorios, escoger aleatoriamente valores que permiten a un programa comportamientos al azar
- **sys:** Proporciona acceso al entorno del sistema operativo. Se considera un módulo avanzado y no se recomienda utilizarlo sin conocimiento.
- **threading:** Divide procesos en subprocesos gracias a hilos de ejecución paralelos.
- **tkinter:** Uno de los módulos de interfaz gráfica más utilizados en Python.

# Módulo collections: colecciones de datos

El módulo integrado de **collections** provee tipos de datos que mejoran los disponibles en Python. **Counter**, por ejemplo, es una subclase de diccionario capaz de realizar cuentas o conteos sobre listas o strings.

## Ejemplo del módulo collections y counter

```
from collections import Counter

numeros = [1,2,3,4,1,2,3,1,2,1]
print(Counter(numeros)) #Counter({1: 4, 2: 3, 3: 2, 4: 1})

print(Counter("palabra"))
#Counter({'a': 3, 'p': 1, 'l': 1, 'b': 1, 'r': 1})
```

Para contar las palabras que hay dentro de una cadena usamos la función **split()**, que separa ese “conjunto de palabras” en una “lista de palabras”.



# Módulo collections: colecciones de datos

## Ejemplo del módulo collections y counter con split()

```
coches = "mercedes ferrari bmw bmw ferrari bmw"
print(Counter(coches.split()))
#Counter({'bmw': 3, 'ferrari': 2, 'mercedes': 1})
animales = "perro gato jirafa jirafa gato jirafa"
a = Counter(animales.split())

# most_common() devuelve una lista ordenada por repeticiones:
print(a.most_common(1)) # Elemento más repetido
print(a.most_common(2)) # Las dos palabras más repetidas
print(a.most_common())  # Ordenado por número de repeticiones
```

## Terminal

```
[('jirafa', 3)]
[('jirafa', 3), ('gato', 2)]
[('jirafa', 3), ('gato', 2), ('perro', 1)]
```

# Módulo datetime: fecha y hora

Para manejar fechas en Python se suele utilizar la librería **datetime** que incorpora los tipos de datos **date**, **time** y **datetime** para representar fechas y métodos para manejarlas. Algunas de las operaciones más habituales que permite son:

- Acceder a los distintos componentes de una fecha (año, mes, día, hora, minutos, segundos y microsegundos).
- Convertir cadenas con formato de fecha en los tipos **date**, **time** o **datetime**.
- Convertir fechas de los tipos **date**, **time** o **datetime** en cadenas formateadas de acuerdo a diferentes formatos de fechas.
- Hacer aritmética de fechas (sumar o restar fechas).
- Comparar fechas.

# Módulo datetime: fecha y hora

## Ejemplo del módulo datetime

```
from datetime import datetime

dt = datetime.now() # Fecha y hora actual

print(dt) # Imprime fecha y hora actual
print("Año:", dt.year) # Año
print("Mes:", dt.month) # Mes
print("Dia:", dt.day) # Dia

print("Hora:", dt.hour) # Hora
print("Minuto:", dt.minute) # Minuto
print("Segundo:", dt.second) # Segundo
print("Microsegundo:", dt.microsecond) # Microsegundo

print("{}: {}: {}".format(dt.hour, dt.minute, dt.second))
print("{} / {} / {}".format(dt.day, dt.month, dt.year))
```

## Terminal

```
2022-11-19 17:00:44.589120
Año: 2022
Mes: 11
Dia: 19

Hora: 17
Minuto: 0
Segundo: 44
Microsegundo: 589120

17:0:44
19/11/2022
```

# Módulo datetime: fecha y hora

Es posible crear un **datetime** pasando parámetros (year, month, day, hour=0, minute=0, second=0, microsecond=0, tzinfo=None). Sólo son obligatorios el año, el mes y el día.

## Objeto datetime

```
from datetime import datetime
dt = datetime(2023,9,28, 11,23)
print(dt) #2023-09-28 11:23:00
```

No se puede cambiar un atributo asignándole un valor porque son de solo lectura. Se debe utilizar el método **replace()**:

## Objeto datetime

```
from datetime import datetime
dt = dt.replace(year=3000)
print(dt) #3000-09-28 11:23:00
```

# Módulo math

Este módulo contiene funciones y métodos para manejar números, hacer redondeos, sumatorias, truncamientos, etcétera:

## Ejemplo del módulo math (I)

```
import math # Importamos el módulo math

print(math.floor(3.99)) # Redondeo a la baja (suelo)
print(math.ceil(3.01)) # Redondeo al alta (techo)

numeros = [0.9999999, 1, 2, 3]
print(math.fsum(numeros)) # 6.9999999

print(math.trunc(123.45)) # 123

print(math.pow(2, 3)) # Potencia con flotante
print(math.sqrt(9)) # Raíz cuadrada (square root)
```

## Terminal

```
3
4

6.9999999

123

8.0
3.0
```

# Módulo math

**math** posee una gran cantidad de constantes útiles, y de funciones que suelen ser necesarias en cualquier proyecto:

## Ejemplo del módulo math (II)

```
import math # Importamos el módulo math

print(math.pi)
print(math.e)

print(math.sin(math.pi / 2)) # Función seno
print(math.cosh(2)) # Función coseno hiperbólico
print(math.radians(30)) # Pasar de grados a radianes
print(math.gcd(12, 34)) # Máximo común divisor
print(math.factorial(7)) # Factorial
# Números complejos (usamos j en lugar de i):
print((3+4j) * (7+10j))
```

## Terminal

```
3.141592653589793
2.718281828459045

1.0
3.7621956910836314
0.5235987755982988
2
5040

(-19+58j)
```

# Módulo random

Este módulo contiene funciones para generar números aleatorios:

## Ejemplo del módulo random

```
import random
# Flotante aleatorio >= 0 y < 1.0
print(random.random())
# Flotante aleatorio >= 1 y <10.0
print(random.uniform(1,10))
# Entero aleatorio de 0 a 9, 10 excluido
print(random.randrange(10))
# Entero aleatorio de 0 a 100
print(random.randrange(0,101))
# Entero aleatorio de 0 a 100 cada 2 números, múltiplos de 2
print(random.randrange(0,101,2))
# Entero aleatorio de 0 a 100 cada 5 números, múltiplos de 5
print(random.randrange(0,101,5))
```

## Terminal

0.4895240624939282

1.2116698460533213

5

65

68

15

# Módulo random

También tiene funciones para tomar muestras:

## Programa

```
# Letra aleatoria
print(random.choice('Hola mundo'))
# Elemento aleatorio
random.choice([1,2,3,4,5])
# Dos elementos aleatorios
random.sample([1,2,3,4,5], 2)
```

## Terminal

a

5

[1, 2]

Y para mezclar colecciones:

## Programa

```
lista = [1,2,3,4,5]
random.shuffle(lista) #Barajar y guardar la lista
print(lista)
```

## Terminal

[2, 4, 3, 1, 5]



# Material extra

# Artículos de interés

Material extra:

- [Excepciones en Python](#)
- [Módulos y paquetes](#)
- [Módulo collections](#)
- [Módulo datetime](#)

Videos:

- [Excepciones](#), en Codigofacilito
- [Módulos en Python](#), de BitBoss

# No te olvides de dar el presente

# Recordá:

- Revisar la Cartelera de Novedades.
- Hacer tus consultas en el Foro.
- Realizar los Ejercicios obligatorios.

**Todo en el Aula Virtual.**