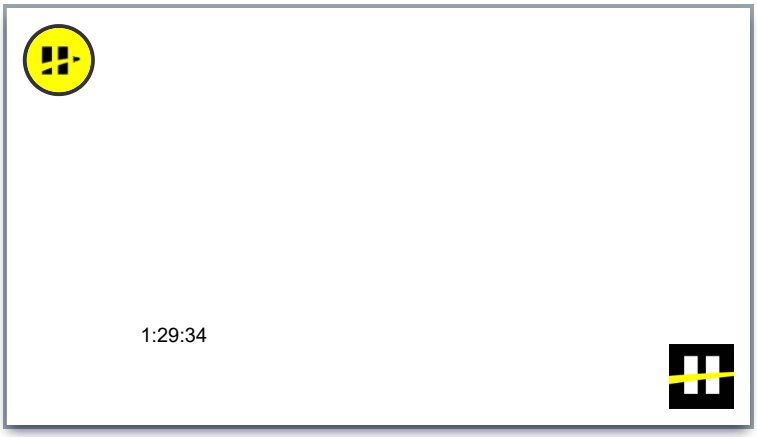
Tiempo de lectura
16 min

**Contenido de la clase**



# Grabación de la Clase 8




# Principales Objetivos de Aprendizaje para esta Clase

- Comprender el concepto de Manejo de Errores
- Conocer las Pruebas de Caja Negra y Caja de Cristal
- Comprender cómo seguir el código paso a paso
- Comprender el concepto de excepciones
- Características de Python

**Dejanos tu feedback!** 👍

# Manejo de Errores

Manejo de Errores | Python - Prep Course

## Contenido de la clase



especificación de la función o el programa, aquí debemos probar sus inputs y validar los outputs. Se llama caja negra por que no necesitamos saber necesariamente los procesos internos del programa, solo contrastar sus resultados.
Hay dos tipos de pruebas muy importantes:

- **Pruebas Unitarias: se realizan pruebas a cada uno de los módulos para determinar su correcto funcionamiento.**
- **Pruebas de Integración: se valida que todos los módulos funcionan entre sí.**

Es una buena práctica realizar las pruebas antes de crear código, esto es porque cualquier cambio que se realice a futuro los test estarán incorporados para determinar si los cambios cumplen lo esperado.
En Python existe la posibilidad de realizar estas pruebas gracias a la librería **unittest**.

```
>>> import unittest
>>>
>>> def suma(num_1, num_2):
>>>     return num_1 + num_2
>>>
>>> class CajaNegraTest(unittest.TestCase):
>>>
>>>     def test_suma_dos_positivos(self):
>>>         num_1 = 10
>>>         num_2 = 5
>>>
>>>         resultado = suma(num_1, num_2)
>>>
>>>         self.assertEqual(resultado, 15)
>>>
>>>     def test_suma_dos_negativos(self):
>>>         num_1 = -10
>>>         num_2 = -7
>>>
>>>         resultado = suma(num_1, num_2)
>>>
>>>         self.assertEqual(resultado, -17)
>>>
>>> unittest.main(argv=[''], verbosity=2, exit=
test_suma_dos_negativos (__main__.CajaNegraTest
test_suma_dos_positivos (__main__.CajaNegraTest
```

**Dejanos tu feedback! 👍**

```
OK
<unittest.main.TestProgram at 0x2226bd08400>
```

## Contenido de la clase



# Pruebas de caja de cristal

Se basan en el flujo del programa, por lo que se asume que conocemos el funcionamiento del programa, por lo que podemos probar todos los caminos posibles de una función. Esto significa que vamos a probar las ramificaciones, bucles for y while, recursiónes, etc.
Este tipo de pruebas son muy buenas cuando descubrimos un bug cuando corremos el programa, por lo que vamos a buscar el **bug** ó error de código gracias a que conocemos su estructura.

```
>>> import unittest

>>> def es_mayor_de_edad(edad):
>>>     if edad >= 18:
>>>         return True
>>>     else:
>>>         return False

>>> class PruebaDeCristalTest(unittest.TestCas
>>>
>>>     def test_es_mayor_de_edad(self):
>>>         edad = 20
>>>
>>>         resultado = es_mayor_de_edad(edad)
>>>
>>>         self.assertEqual(resultado, True)
>>>
>>>     def test_es_menor_de_edad(self):
>>>         edad = 15
>>>
>>>         resultado = es_mayor_de_edad(edad)
>>>
>>>         self.assertEqual(resultado, False)

>>> unittest.main(argv=[''], verbosity=2, exit=
test_es_mayor_de_edad (__main__.PruebaDeCrista
test_es_menor_de_edad (__main__.PruebaDeCrista
```

**Dejanos tu feedback! 👍**

```
OK
<unittest.main.TestProgram at 0x2226bd153d0>
```

## Contenido de la clase



# Seguir el código paso a paso ó 'Debugging'

Los bugs son un problema que les sucede a todos, sin embargo si realizamos test a nuestro programa probablemente tendremos menos bugs, pero esto no es suficiente.
Existen unas reglas generales que nos ayudaran:

- **Aprende a utilizar el print statement.**
- **Estudia los datos disponibles.**
- **Utiliza los datos para crear hipótesis y experimentos. Método científico.**
- **Ten una mente abierta. Si entendes el programa, probablemente no habrán bugs.**
- **Lleva un registro de lo que has tratado, preferentemente en la forma de tests.**
- **Debuguear es un proceso de búsqueda de los bugs, por lo que al diseñar nuestros experimentos debemos acotar el espacio de búsqueda en cada prueba. Una forma ágil de debugear es utilizando una búsqueda binaria con print statements, esto significa que ejecutamos la mitad del código, si no falla entonces sabemos que el problema está en la otra mitad, y en cada área que vamos acortando lo dividimos por mitades, de esta forma hallaremos rápidamente nuestro bug.**

Existe un listado de errores comunes de los cuales también nos podemos apoyar:

- **Encuentra a los sospechosos comunes (llamado a una función mal escrita, parámetros en orden incorrecto, etc.)**
- **En lugar de preguntarte por qué un programa no funciona, pregúntate por qué está funcionando de esta manera.**
- **Es posible que el bug no se encuentre donde crees que está.**

**Dejanos tu feedback! 👍**

**preferentemente en la forma de tests.**

## Contenido de la clase



# Manejo de excepciones

Los manejos de excepciones son muy comunes en la programación, no tienen nada de excepcional. Las excepciones de Python normalmente se relacionan con errores de semántica, también podemos crear nuestras propias excepciones, pero cuando una excepción no se maneja (unhandled exception), el programa termina en error.

Las excepciones se manejan con los keywords: **try**, **except**, **finally**. Se pueden utilizar también para ramificar programas.
No deben manejarse de manera silenciosa (por ejemplo, con print statements). Para crear tu propia excepción utiliza el keyword **raise**.

```
>>> def divide_elementos_de_lista(lista, divis
>>>     '''
>>>     Cada elemento de una lista es dividida
>>>     En caso de error de tipo ZeroDivisionEr
>>>     significa error al dividir en cero
>>>     la función devuelve la lista inicial
>>>     '''
>>>     try:
>>>         return [i / divisor for i in lista
>>>
>>>     except ZeroDivisionError as e:
>>>         print(e)
>>>         return lista
>>>
>>> lista = list(range(10))
>>> divisor = 0
>>>
>>> print(divide_elementos_de_lista(lista, div
division by zero
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> divisor = 3
>>> print(divide_elementos_de_lista(lista, div
[0.0, 0.3333333333333, 0.6666666666666, 1.0, 1
```

# Excepciones y control de flujo

**Dejanos tu feedback! 👍**

Manejo de Errores | Python - Prep Course

## Contenido de la clase



de flujo.

¿Por qué es necesaria otra modalidad para controlar el flujo? Una razón muy específica: el principio EAFP (easier to ask for forgiveness than permission, es más fácil pedir perdón que permiso, por sus siglas en inglés).

El principio EAFP es un estilo de programación común en Python en el cual se asumen llaves, índices o atributos válidos y se captura la excepción si la suposición resulta ser falsa. Es importante resaltar que otros lenguajes de programación favorecen el principio LBYL (look before you leap, revisa antes de saltar) en el cual el código verifica de manera explícita las precondiciones antes de realizar llamadas.

```python
#Python
def busca_pais(paises, pais):
'''
Paises es un diccionario. Pais es la llave.
Codigo con el principio EAFP.
'''

try:
    return paises[pais]
except KeyError:
     return None
```

```javascript
// Javascript

/**
* Paises es un objeto. Pais es la llave.
* Codigo con el principio LBYL.
*/
function buscaPais(paises, pais) {
  if(!Object.keys(paises).includes(pais)) {
    return null;
  }

  return paises[pais];
}
```

**Dejanos tu feedback! 👍**

Manejo de Errores | Python - Prep Course

JavaScript, se verifica primero que la llave exista en el objeto y únicamente con posterioridad se accede.

Es importante resaltar que ambos estilos pueden utilizarse en Python, pero el estilo EAFP es mucho más propio de este lenguaje.

```
#All possible errors

except TypeError:
    print("is thrown when an operation or funct
except IndexError:
    print("is thrown when trying to access an i
except KeyError:
    print("is thrown when a key is not found.")
except ImportError:
    print("Raised when the imported module is r
except StopIteration:
    print("is thrown when the next() function g
except ValueError:
    print("is thrown when a function's argument
except NameError:
    print("is thrown when an object could not b
except ZeroDivisionError:
    print("is thrown when the second operator i
except KeyboardInterrupt:
    print("is thrown when the user hits the int
except MemoryError:
    print("Raised when an operation runs out of
except FloatingPointError:
    print("Raised when a floating point operati
except OverflowError:
    print("Raised when the result of an arithme
except ReferenceError:
    print("Raised when a weak reference proxy i
except TabError:
    print("Raised when the indentation consists
except SystemError:
    print("Raised when the interpreter detects
except RuntimeError:
    print("Raised when an error does not fall u
except:
    print("Error detected can't be handled nor
```

## Contenido de la clase



**Dejanos tu feedback!** 👍

## Contenido de la clase



podemos determinar si una afirmación se cumple o no se cumple y poder seguir adelante con la ejecución de nuestro programa o darle término.
Consiste en un método de programación defensiva, esto significa que nos estamos preparando para verificar que los tipos de inputs de nuestro programa es del tipo que nosotros esperamos. Estos también nos sirven para debuggear.
Para realizar una afirmación en nuestro programa lo hacemos con la expresión assert (expresion booleana), (mensaje de error).

```
>>> def primera_letra(lista_de_palabras):
>>>   primeras_letras = []
>>>
>>>   for palabra in lista_de_palabras:
>>>     assert type(palabra) == str, f'{palabra
>>>     assert len(palabra) > 0, 'No se permite
>>>
>>>     primeras_letras.append(palabra[0])
>>>   return primeras_letras
```

# Características de Python

- **Es un lenguaje interpretado, no compilado.**
- **Usa tipado dinámico, lo que significa que una variable puede tomar valores de distinto tipo.**
- **Es fuertemente tipado, lo que significa que el tipo no cambia de manera repentina. Para que se produzca un cambio de tipo tiene que hacer una conversión explícita.**
- **Es multiplataforma, ya que un código escrito en macOS funciona en Windows o Linux y viceversa.**

**Dejanos tu feedback! 👍**

# Homework

## Contenido de la clase



Completa la tarea descrita en el archivo **README**

Si tienes dudas sobre este tema, puedes consultarlas en el canal #python de Slack

Hecho con 💛 por alumnos de Henry

**Dejanos tu feedback! 👍**