DESCUBRIENDO EL PODER DE LA PROGRAMACION

CURSO INICIAL DE PYTHON

Jimmy Chung
Alexander Solis

CONTENIDO DEL CURSO

1. Introducción, Instalación y Conceptos básicos
2. Variables, Expresiones, Funciones y Operadores
3. Condicionales y Ciclos – Patrones de sintaxis válidos
4. Listas, tuplas y diccionarios
5. Errores y Excepciones
6. Objetos, Clases y funciones en Python
7. Guardar y Recuperar datos de archivos

Calse 5: Errores y Excepciones

Funciones «built-in»

Funciones «built-in»: Incorporadas por defecto en el propio lenguaje

`abs()`	`delattr()`	`hash()`	`memoryview()`	`set()`
`all()`	`dict()`	`help()`	`min()`	`setattr()`
`any()`	`any()`	`hex()`	`next()`	`slice()`
`ascii()`	`divmod()`	`id()`	`object()`	`sorted()`
`bin()`	`enumerate()`	`input()`	`oct()`	`staticmethod()`
`bool()`	`eval()`	`int()`	`open()`	`str()`
`breakpoint()`	`exec()`	`isinstance()`	`ord()`	`sum()`
`bytearray()`	`filter()`	`issubclass()`	`pow()`	`super()`
`bytes()`	`float()`	`iter()`	`print()`	`tuple()`
`callable()`	`format()`	`len()`	`property()`	`type()`
`chr()`	`frozenset()`	`list()`	`range()`	`vars()`
`classmethod()`	`getattr()`	`locals()`	`repr()`	`zip()`
`compile()`	`globals()`	`map()`	`reversed()`	`__import__()`
`complex()`	`hasattr()`	`max()`	`round()`	

https://docs.python.org/es/3/library/functions.html?highlight=built

Excepciones y Errores

La gestión de excepciones y errores es una parte fundamental de la programación, ya que nos permite manejar situaciones inesperadas que pueden ocurrir durante la ejecución de un programa. Python proporciona una estructura de manejo de excepciones que nos permite capturar y manejar los errores de manera controlada.

La sintaxis básica para el manejo de excepciones en Python es la siguiente:

```
try:
    # Código que puede generar una excepción
    # ...
except TipoDeExcepcion:
    # Código para manejar la excepción
    # ...
```

Excepciones y Errores

Para capturar un error excepción utilizamos la sentencia try:

```
try:
a = 4
b = 0
print(a/b)
print(”Fin del programa”)
# ZeroDivisionError: division by zero
# Si nuestro programa tiene mas
instrucciones, estas no se ejecutarán,
Ejemplo (print(”Fin del programa”))
```

```
try:
    a = 4
    b = 0
    print(a/b)
except:
    print(”Ocurrió un error”)

print(”Fin del programa”)
```

Excepciones más comunes

Todas estas excepciones heredan de la clase Exception.

Nosotros podemos crear nuestras propias excepciones, las cuales tienen que heredar de la clase Exception.

SyntaxError: Ocurre cuando el intérprete de Python encuentra un error de sintaxis en el código fuente, como una palabra clave mal escrita, un paréntesis faltante o una indentación incorrecta.

IndentationError: Se produce cuando hay un error en la indentación del código, como mezclar espacios y tabulaciones o no mantener una indentación coherente.

https://docs.python.org/es/3/library/exceptions.html#concrete-exceptions

Excepciones más comunes

NameError: Sucede cuando se intenta utilizar una variable o un nombre que no ha sido definido previamente en el programa.

TypeError: Se produce cuando se realiza una operación o se utiliza una función con un tipo de dato incorrecto o incompatible. Por ejemplo, intentar sumar una cadena de texto con un número.

IndexError: Ocurre cuando se intenta acceder a un índice fuera del rango válido en una lista, tupla o cadena de texto.

KeyError: Sucede cuando se intenta acceder a una clave que no existe en un diccionario.

https://docs.python.org/es/3/library/exceptions.html#concrete-exceptions

Excepciones más comunes

ValueError: Se produce cuando se llama a una función con un argumento de valor incorrecto, como intentar convertir una cadena de texto no numérica a un número.

FileNotFoundError: Ocurre cuando se intenta abrir o acceder a un archivo que no existe en el sistema.

ImportError: Sucede cuando hay un error al importar un módulo o una biblioteca. Puede ocurrir si el módulo no está instalado o si la ruta de importación es incorrecta.

ZeroDivisionError: Ocurre cuando se intenta dividir un número por cero.

https://docs.python.org/es/3/library/exceptions.html#concrete-exceptions

Excepciones y Errores: finally y else

Además del bloque except, también se pueden utilizar los bloques else y finally.

El bloque else: Se ejecuta cuando no ocurre ninguna excepción en el bloque try. Es útil para ejecutar código adicional cuando no se produce ningún error.

El bloque finally: Se ejecuta siempre, independientemente de si se produjo una excepción o no.

```
try:
    numero1 = int(input("Ingrese un número: "))
    numero2 = int(input("Ingrese otro número: "))

    resultado = numero1 / numero2

    print("El resultado de la división es:", resultado)

except ValueError:
    print("Error: Debe ingresar un número válido.")

except ZeroDivisionError:
    print("Error: No se puede dividir entre cero.")

except Exception as e:
    print("Error:", str(e))

else:
    print("La división se realizó correctamente.")

finally:
    print("Fin del programa.")
```

Ejercicios

- División por cero:

```
try:
    resultado = 10 / 0
except ZeroDivisionError:
    print("Error: División por cero")
```

```
def dividir(a, b):
    if b == 0:
        raise ValueError("Divisor no puede ser cero")
    return a / b

try:
    resultado = dividir(10, 0)
except ValueError as error:
    print("Error:", str(error))
```

Ejercicios

- Iteración sobre un objeto no iterable:

```
try:
    for elemento in 10:
        print(elemento)
except Exception as error:
    print(error)
```

```
try:
    for elemento in 10:
        print(elemento)
except TypeError:
    print("Error: Objeto no iterable")
```

- Acceso a una clave inexistente en un diccionario:

```
diccionario = {"nombre": "Juan", "edad": 30}
try:
    valor = diccionario["apellido"]
except KeyError:
    print("Error: Clave inexistente")
```

Ejercicios

- Uso incorrecto de un módulo:

```
try:
    import modulo_inexistente
except ImportError:
    print("Error: Módulo inexistente")
```

- Apertura de un archivo inexistente:

```
try:
    archivo = open("archivo.txt", "r")
except FileNotFoundError:
    print("Error: Archivo no encontrado")
```

Ejercicios

- Conversión de tipo inválida:

```
lista = [1, 2, 3]
try:
    elemento = lista[3]
except IndexError:
    print("Error: Índice fuera de rango")
```

- Acceso a índices fuera de rango:

```
try:
    numero = int("abc")
except ValueError:
    print("Error: Conversión de tipo inválida")
```

Calse 5: Objetos, Clases y Funciones

POO: Objetos, Clases y Funciones

Las clases y funciones son dos elementos fundamentales en la programación orientada a objetos (POO) en Python.

FUNCIONES: Las funciones en Python son bloques de código que se pueden llamar para realizar una tarea específica. Pueden tomar cero o más argumentos y pueden devolver un valor de salida. Las funciones se definen utilizando la palabra clave def, seguida del nombre de la función y los parámetros entre paréntesis.

Dentro de una función, se pueden realizar operaciones y cálculos, y se puede utilizar la palabra clave return para devolver un valor. Las funciones son una forma de encapsular la lógica del programa y hacerlo más modular y fácil de mantener.

POO: Objetos, Clases y Funciones

CLASES: Una clase es una plantilla o molde para crear objetos. Define las características y comportamientos que los objetos de esa clase tendrán. Las clases en Python se definen utilizando la palabra clave class, seguida del nombre de la clase. Dentro de una clase, puedes definir atributos (variables) y métodos (funciones) que operan en esos atributos. Los objetos creados a partir de una clase se llaman instancias y pueden tener valores únicos para sus atributos. Las clases son una forma de organizar y estructurar el código de manera modular y reutilizable.

Programación Orientada a Objetos (POO)

OBJETO: Un objeto es una entidad que representa algo del mundo real en el contexto de la programación. Puede ser cualquier cosa, como una persona, un coche, un libro, etc. Los objetos tienen características (llamadas atributos) que describen su estado y comportamientos (llamados métodos) que indican lo que pueden hacer.

Imagine que tiene una clase llamada "Carro", puede crear varios objetos de esa clase, como un carro rojo, un carro azul y un carro negro. Todos estos objetos son carros, pero tienen diferentes atributos, como el color, el modelo, la marca, entre uchos otros atributos.

Programación Orientada a Objetos (POO)

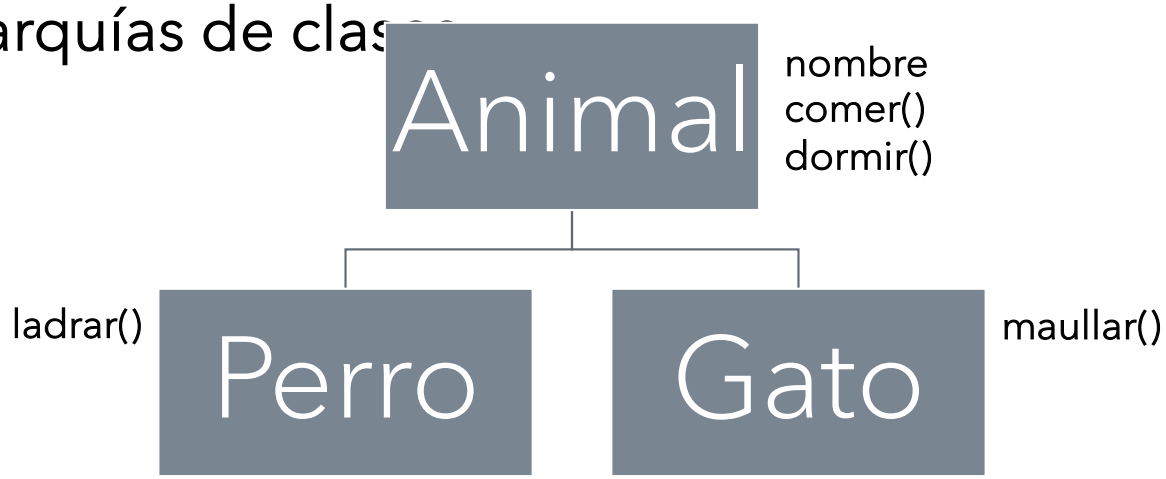
La programación orientada a objetos (POO) es un paradigma de programación que se basa en el concepto de "objetos". Un objeto es una entidad que tiene atributos y comportamientos. En POO, se crean clases que son plantillas o moldes para crear objetos.

La programación orientada a objetos permite representar entidades del mundo real de manera más intuitiva, proporciona una forma eficiente de gestionar y manipular datos, nos permite organizar el código en clases y objetos. Todo esto facilita la creación de programas estructurados, modulares y reutilizables:

Programación Orientada a Objetos (POO)

La programación orientada a objetos se basa en cuatro conceptos fundamentales:

Herencia: Permite crear nuevas clases basadas en clases existentes. La herencia permite que una clase herede atributos y métodos de otra clase, lo que facilita la reutilización de código y la creación de jerarquías de clas




```
mi_perro.comer()
mi_gato.dormir()
mi_perro.ladrar()
mi_gato.maullar()
```

Programación Orientada a Objetos (POO)

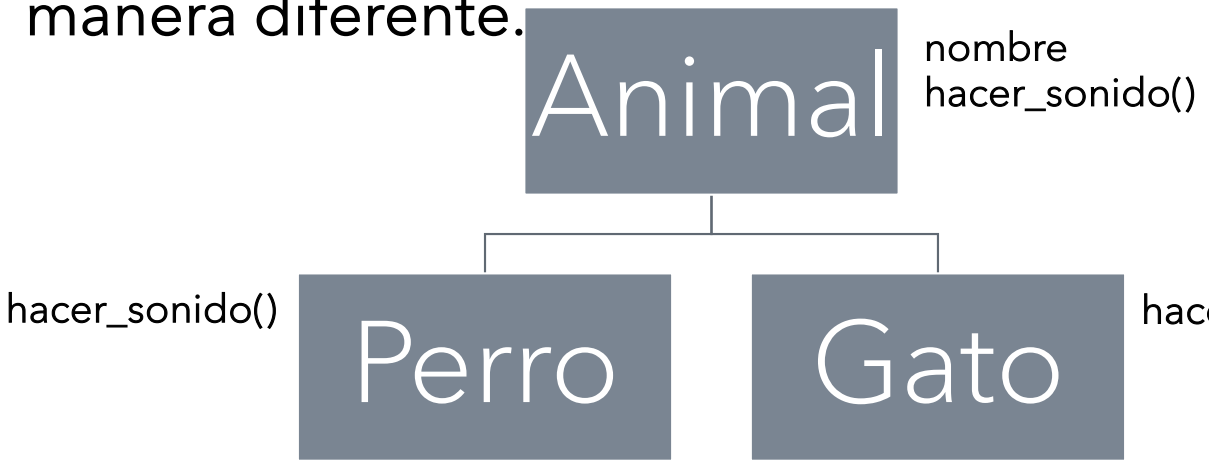
- POO: Herencia:

```python
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def comer(self):
        print(f"{self.nombre} está comiendo.")

    def dormir(self):
        print(f"{self.nombre} está durmiendo.")

class Perro(Animal):
    def ladrar(self):
        print("¡Guau, guau!")

class Gato(Animal):
    def maullar(self):
        print("¡Miau, miau!")
```

```python
# Crear objetos de las clases derivadas
mi_perro = Perro("Firulais")
mi_gato = Gato("Michi")

# Acceder a los métodos de la clase base y las clases derivadas
mi_perro.comer()  # Salida: Firulais está comiendo.
mi_gato.dormir()  # Salida: Michi está durmiendo.
mi_perro.ladrar()  # Salida: ¡Guau, guau!
mi_gato.maullar()  # Salida: ¡Miau, miau!
```

Programación Orientada a Objetos (POO)

Polimorfismo: Es la capacidad de un objeto para tomar diferentes formas o comportarse de diferentes maneras según el contexto. En POO, se puede utilizar el polimorfismo para que objetos de diferentes clases respondan a un mismo mensaje o método de manera diferente.




```
def hacer_sonar(animal):

print(animal.hacer_sonido())


hacer_sonar(mi_perro)
hacer_sonar(mi_gato)
```

Programación Orientada a Objetos (POO)

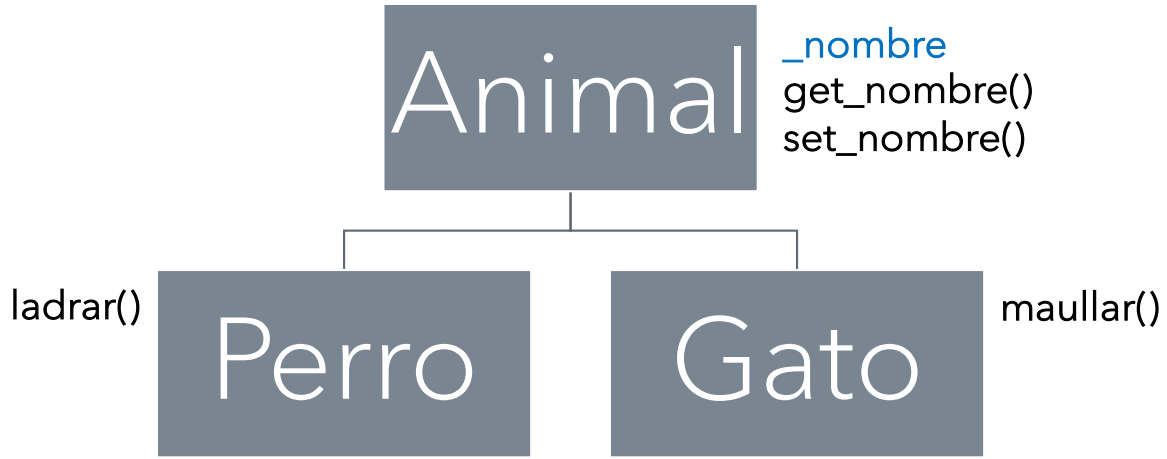
- POO: Polimorfismo:

```python
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        return "¡Guau, guau!"

class Gato(Animal):
    def hacer_sonido(self):
        return "¡Miau, miau!"

# Función que hace sonar a un animal
def hacer_sonar(animal):
    print(animal.hacer_sonido())
```

```python
# Crear objetos de las clases derivadas
mi_perro = Perro("Firulais")
mi_gato = Gato("Michi")

# Llamar a la función con diferentes objetos
hacer_sonar(mi_perro)  # Salida: ¡Guau, guau!
hacer_sonar(mi_gato)  # Salida: ¡Miau, miau!
```

Programación Orientada a Objetos (POO)

Encapsulación: Es el proceso de ocultar los detalles internos de un objeto y proporcionar una interfaz para interactuar con él. Los atributos y métodos internos de un objeto pueden estar ocultos y solo se accede a ellos a través de métodos públicos.




```
mi_perro.get_nombre()
mi_gato. get_nombre()
```

Programación Orientada a Objetos (POO)

- POO: Encapsulación:

```python
class Animal:
    def __init__(self, nombre):
        self._nombre = nombre

    def get_nombre(self):
        return self._nombre

    def set_nombre(self, nuevo_nombre):
        self._nombre = nuevo_nombre

class Perro(Animal):
    def hacer_sonido(self):
        return "¡Guau, guau!"

class Gato(Animal):
    def hacer_sonido(self):
        return "¡Miau, miau!"
```

```python
# Crear objetos de las clases derivadas
mi_perro = Perro("Firulais")
mi_gato = Gato("Michi")

# Acceder al nombre del perro
print(mi_perro.get_nombre())  # Salida: Firulais

# Cambiar el nombre del perro
mi_perro.set_nombre("Max")

# Acceder al nombre actualizado del perro
print(mi_perro.get_nombre())  # Salida: Max
```

Programación Orientada a Objetos (POO)

Abstracción: Es el proceso de simplificar un objeto o una clase al identificar solo las características y comportamientos esenciales. La abstracción permite enfocarse en los aspectos importantes y ocultar los detalles innecesarios.

Un concepto relacionado con la abstracción, serían las clases abstractas o más bien los métodos abstractos. Se define como clase abstracta la que contiene métodos abstractos, y se define como método abstracto a un método que ha sido declarado pero no implementado. Es decir, que no tiene código.

Es posible crear métodos abstractos en Python con el decorador @absttractmethod

Programación Orientada a Objetos (POO)

- POO: Abstracción:

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, nombre):
        self.nombre = nombre

    @abstractmethod
    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        return "¡Guau, guau!"

class Gato(Animal):
    def hacer_sonido(self):
        return "¡Miau, miau!"
```

```python
# No se puede crear un objeto de la clase Animal directamente
# animal = Animal("Animal")  # Generaría un error

# Crear objetos de las clases derivadas
perro = Perro("Firulais")
gato = Gato("Michi")

# Llamar al método hacer_sonido de cada objeto
print(perro.hacer_sonido())  # Salida: ¡Guau, guau!
print(gato.hacer_sonido())  # Salida: ¡Miau, miau!
```

Clase 5: Guardar y Recuperar datos de archivos

Archivos de texto: Abrir archivos

Python proporciona varias funciones y métodos integrados para leer y escribir archivos de texto.

Para trabajar con archivos de texto, primero debemos abrir el archivo utilizando la función open(). Esta función toma dos argumentos: el nombre del archivo y el modo de apertura. El modo de apertura puede ser "r" (lectura), "w" (escritura) o "a" (agregar):

```
archivo = open("ejemplo.txt", "r")
```

Una vez que tenemos el archivo abierto, podemos leer su contenido o escribir datos en él.

Archivos de texto: Leer contenido

Para leer el contenido de un archivo, podemos utilizar el método read(). Este método lee todo el contenido del archivo y lo devuelve como una cadena de caracteres.

```
contenido = archivo.read()
print(contenido)
```

Para leer un archivo línea por línea, se utiliza el método readline()

```
for linea in archivo:
    print(linea)
```

Archivos de texto: Escribir datos

Para escribir en un archivo, podemos utilizar el método write(). Este método toma una cadena de caracteres como argumento y escribe la cadena en el archivo.

```
archivo.write("Hola, mundo!")
```

Es importante recordar cerrar el archivo después de terminar de trabajar con él utilizando el método close(). Esto libera los recursos asociados al archivo.

```
archivo.close()
```

Archivos de texto: Escribir datos

Python también proporciona una forma más conveniente y segura de trabajar con archivos de texto utilizando el bloque with. El bloque with se encarga automáticamente de cerrar el archivo una vez que se termina de trabajar con él.

```
with open("ejemplo.txt", "r") as archivo:
    contenido = archivo.read()
    print(contenido)
```

Referencias bibliográficas

Documentación oficial Python.

https://docs.python.org/es/3/tutorial/index.html

https://docs.python.org/es/3/library/exceptions.html#exception-context

https://docs.python.org/es/3/library/exceptions.html#concrete-exceptions

https://docs.python.org/es/3/library/functions.html#open

https://docs.python.org/es/3/glossary.html#term-file-object