

#

DESARROLLO WEB FULLSTACK

con Python y JavaScript

polotic
misiones



Excepciones

Tratamiento de Excepciones

Como vimos en clases anteriores, pueden ocurrir errores en Python que se llaman Excepciones. Estas excepciones pueden ser **esperables** y pueden ser **tratables** para que continúe el funcionamiento del sistema.

En el siguiente fragmento de código, tomaremos dos números enteros del usuario e intentaremos dividirlos:

```
numero1 = int(input("Ingrese primer numero: "))
numero2 = int(input("Ingrese segundo numero: "))

resultado = numero1 / numero2

print(f"{numero1} / {numero2} = {resultado}")
```


Tratamiento de Excepciones

Este código puede ofrecer dos respuestas

```
numero1 = int(input("Ingrese primer numero: "))
numero2 = int(input("Ingrese segundo numero: "))

resultado = numero1 / numero2

print(f"{numero1} / {numero2} = {resultado}")
```



```
Ingrese primer numero: 5
Ingrese segundo numero: 2
5 / 2 = 2.5
>
```

```
Ingrese primer numero: 2
Ingrese segundo numero: 0
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    resultado = numero1 / numero2
ZeroDivisionError: division by zero
>
```

try ... except ...

- Podemos lidiar con este error usando Manejo de Excepciones.
- En el siguiente bloque de código, intentaremos (try) dividir los dos números, excepto (except) cuando obtengamos un ZeroDivisionError:

```
import sys

numero1 = int(input("Ingrese primer numero: "))
numero2 = int(input("Ingrese segundo numero: "))
try:
    resultado = numero1 / numero2
except ZeroDivisionError:
    print("Error: No se puede dividir por 0.")
    #Salir del programa
    sys.exit(1)

print(f"{numero1} / {numero2} = {resultado}")
```

```
Ingrese primer numero: 5
Ingrese segundo numero: 0
Error: No se puede dividir por 0.
repl process died unexpectedly: exit status 1
```

Tratamiento de Excepciones

- Sin embargo aún tenemos errores si el usuario ingresa caracteres que no son números:

```
Ingrese primer numero: c
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    numerol = int(input("Ingrese primer numero: "))
ValueError: invalid literal for int() with base 10: 'c'
➤
```

- Tendremos que controlar también el ingreso de datos al sistema con otro try

Multiples tipos de Excepciones

```
import sys

try:
    numero1 = int(input("Ingrese primer numero: "))
    numero2 = int(input("Ingrese segundo numero: "))
except ValueError:
    print("Error: Valor no valido. ")
    sys.exit(1)

try:
    resultado = numero1 / numero2
except ZeroDivisionError:
    print("Error: No se puede dividir por 0.")
    #Salir del programa
    sys.exit(1)

print(f"{numero1} / {numero2} = {resultado}")
```

```
Ingrese primer numero: c
Error: Valor no valido.
repl process died unexpectedly: exit status 1
```

Tratamiento de Excepciones

else

- Puedes usar la palabra clave `else` para definir un bloque de código que se ejecutará si no se produjeron errores:

```
try:
    print("Hola")
except:
    print("Algo salió mal")
else:
    print("Nada salió mal")
```


Tratamiento de Excepciones

finally

- El bloque finally, si se especifica, se ejecutará independientemente de si el bloque try genera un error o no.

```
try:
    print("Hola")
except:
    print("Algo salió mal")
finally:
    print("El try y except finalizó")
```

Tratamiento de Excepciones

finally

- Ejemplo intentando abrir y escribir a un archivo que es de solo lectura:

```
try:
    f = open("archivoejemplo.txt")
    f.write("Linea de prueba dentro del archivo.")
except:
    print("Algo pasó al intentar abrir el archivo.")
finally:
    f.close()
```



Programación Orientada a Objetos



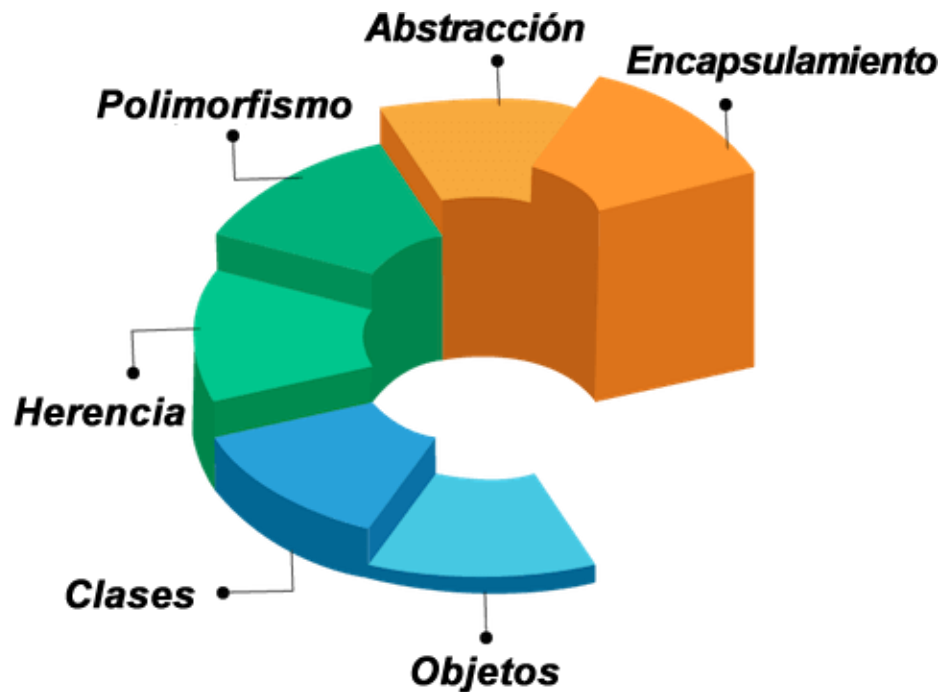
¿Qué es Programación Orientada a Objetos?



La programación orientada a objetos es un **paradigma de programación**, o una forma de pensar sobre la programación, que **se centra en objetos** que pueden almacenar información y realizar acciones.

- Hace que el desarrollo de grandes proyectos de software sea mas fácil y mas intuitivo.
- Nos permite pensar sobre el software en términos de **objetos del mundo real** y sus relaciones.

Propiedades de la POO





Enlaces de Interés

Enlaces de interés:

- Wikipedia: https://es.wikipedia.org/wiki/Programación_orientada_a_objetos
- MIT: <https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-01sc-introduction-to-electrical-engineering-and-computer-science-i-spring-2011/unit-1-software-engineering/object-oriented-programming/>
- Brown University: <http://cs.brown.edu/courses/cs015/>
- Princeton: <https://introcs.cs.princeton.edu/java/30oop/>



Objetos

- Una entidad o “cosa” en tu programa, usualmente un sustantivo.
- Un ejemplo de un objeto sería una persona en particular.

Propiedades

- ☐ Nombre
- ☐ Edad
- ☐ Dirección

Comportamiento

- ☐ Caminar
- ☐ Hablar
- ☐ Respirar

Programación Orientada a Objetos

CLASE:
Vehículo

OBJETO:
Automóvil

Atributos/Propiedades:
marca
modelo
color

Métodos:
inyectar_combustible
frenar
cambiar_marcha



Clases

- Las clases se usan para crear objetos
- Podemos crear muchos objetos desde una sola clase
- Las Clases definen el tipo de datos (**type**)
- El proceso de crear un objeto desde una clase se denomina **instanciación**.

Aquí, creamos una nueva variable denominada `mi_nombre` con el valor de “Matias”. Ésta variable es en realidad una referencia a un **objeto**. El tipo de objeto es **str** porque para poder crearla, necesitamos **instanciar** desde la clase **str**.

```
mi_nombre = “Matias”
```

Ejemplo: Puertas

Clase Puerta

Propiedades:

altura
color
esta_bloqueada

Comportamientos

abrir()
cerrar()
activar_cerradura()

puerta1

altura: 85
color: rojo
esta_bloqueada: False

abrir()
cerrar()
activar_cerradura()

puerta2

altura: 95
color: gris
esta_bloqueada: True

abrir()
cerrar()
activar_cerradura()



Objetos únicos de tipo **Puerta**

Programación Orientada a Objetos

- ❖ **Clases:** ya hemos visto algunos tipos diferentes de variables en Python, pero *¿qué pasa si queremos crear nuestro propio tipo?*
- ❖ Una clase de Python es esencialmente **una plantilla para un nuevo tipo de objeto** que puede almacenar información y realizar acciones. Creemos nuestra primer clase:

```
class Perro:  
    pass
```

El nombre de la clase siempre se escribe en notación de palabras en mayúscula. Como por ejemplo: EstaEsMiNuevaClase

Atributos de Instancia

Agreguemos propiedades que todos los perros deberían tener. Para hacer las cosas mas simples solo tomamos dos. Las propiedades de todos los objetos de tipo Perro deben definirse en un método denominado `__init__()`

- `__init__()` define el **estado** inicial del objeto asignando valores a las propiedades del objeto.
- Esto significa que `__init__()` inicializa cada nueva instancia de la clase.
- Le puedes pasar la cantidad de parámetros que quieras, siempre y cuando el primero sea `self`.
- Recuerda que los métodos se declaran de manera similar a las funciones con `def` como primer elemento.

```
class Perro:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
```

Atributos de Clase

Por otro lado, los atributos de clase son atributos que tienen el mismo valor para todas las instancias. Puedes definir atributos de clase declarándolos por fuera del método `__init__()`

Por ejemplo, en la misma clase Perro podemos asignar un atributo especie que sea siempre la misma en todos:

```
class Perro:
    #Atributo de clase
    especie = "Canis lupus familiaris"
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
```

Los atributos de clase siempre se encontrarán directamente debajo de la definición del nombre de la clase.



Instancias



Puedes instanciar rápidamente una clase en el interprete de Python tipeando el nombre de la clase con paréntesis:

```
>>> Perro()
```

Esto hace que la instancia de Perro se coloque en una posición de memoria. Para poder almacenarla en una variable hacemos lo siguiente:

```
>>> perro1 = Perro()
```

```
>>> perro2 = Perro()
```

Todo esto funciona de maravillas en nuestra clase Perro vacía. No en la que hicimos con atributos en el constructor.

Instancias

Para instanciar el perro con atributos de nombre y edad hacemos lo siguiente:

```
>>> perro1 = Perro("Firulaïs", 8)
```

```
>>> perro2 = Perro("Napoleón", 5)
```

Obviando el parámetro `self` ya que es transparente al usuario en Python.

Puedes acceder a cada uno de los atributos de cada instancia mediante el nombre de la variable y el nombre del atributo asociado (que hayas definido en la clase)

```
>>> perro1.edad
```

```
>>> perro2.nombre
```

```
>>> perro2.especie
```

Instancias

Imaginate que querramos cambiar la especie del perro:

```
>>> perro1.especie = "Felix silvestris"
```

Felix silvestris es el nombre de la especie asociada a los gatos. Esto hace que nuestro perro1 sea un perro bastante extraño. Sin embargo en Python esto es valido.

Metodos de Instancias

Los métodos de instancia son funciones definidas dentro de una clase que solo pueden ser llamadas desde la instancia de la clase. Como el método `__init__()`, en un método de instancia siempre el primer parámetro será `self`.

```
class Perro:

    #Atributo de clase
    especie = "Canis lupus familiaris"
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    #Metodo de instancia
    def descripcion(self):
        return f"{self.nombre} tiene {self.edad} años"

    #Otro metodo de instancia
    def ladrar(self, sonido):
        return f"{self.nombre} dice {sonido}"
```

¿Cómo usamos los métodos?

Para llamar a cada método simplemente utilizamos el operador unario (el punto) y entre paréntesis pasamos los parámetros (que puede o no tener):

```
>>> perro1 = Perro("Fatiga", 12)
```

```
>>> perro1.descripcion()
```

```
>>> perro1.ladrar("Guau guau")
```

```
>>> perro1.ladrar("Aquí me pongo a cantar, al compás de la vigüela...")
```

Hacer nuestra instancia comprensible



Si llamamos a nuestra instancia perro1 mediante `print(perro1)` veremos información sobre la misma que no es clara para el ojo humano inexperto. Entonces estaría bueno poder agregar información relacionada con el significado que le hemos dado previamente. Eso lo hacemos con el método `__str__()`. Tanto el método `__init__` como el `__str__` se denominan métodos Dunder o Magicos en Python y se escriben entre dobles guiones bajos.

Dunder por *Double Underscores* (Dobles Guiones Bajos).

```
class Perro:
    #Todo el resto del código anterior iría aquí que por
    #cuestiones de espacio no lo coloco

    #Se puede reemplazar el método descripcion() con __str__()
    def __str__(self):
        return f"{self.nombre} tiene {self.edad} años"
```

Ahora al realizar `print(perro1)` con el perro1 ya instanciado obviamente, obtendremos la descripción del mismo.

Funcionalidades “built in”



Cuando creas una clase, puedes pensar fácilmente que todo lo que obtienes es la clase. Sin embargo, Python agrega funcionalidad incorporada (o built in) a tu clase. Por ejemplo, en la sección anterior, escribimos `__class__` y vimos que el atributo `__class__` está integrado; vos no lo creaste.

Generalmente las funcionalidades que mas se requieren en todas las clases son incorporadas por defecto.

Si tipeamos: `dir(MiInstancia)`

Veremos el listado de funcionalidades incorporadas.

Puedes obtener mas ayuda con: `help('__class__')`

¡Cuidado con las variables de clase!

Debes recordar que cualquier modificación al valor de una variable de clase es arrastrada hacia las instancias. Ej.:

```
class MiClase:
    Saludo = ""

    def DecirHola(self):
        print("Hola {}".format(self.Saludo))

>> MiClase.Saludo = "Harry"
>> MiClase.Saludo
'Harry'
>> MiInstancia = MiClase()
>> MiInstancia.DecirHola()
'Hola Harry'
```

- Evita las variables de clase cuando puedas porque son intrínsecamente inseguras.
- Siempre inicializa las variables de clase a un valor conocido en el código del constructor.

Metodos con argumento variables

A veces, necesitas crear métodos que puedan tomar un número variable de argumentos. Manejar este tipo de situaciones es algo que Python hace bien. Estos son los dos tipos de argumentos variables que puedes crear:

- ✓ `* args`: proporciona una lista de argumentos sin nombre.
- ✓ `** kwargs`: proporciona una lista de argumentos con nombre.

```
class MiClase:
    def ImprimirLista1(*args):
        for Count, Item in enumerate(args):
            print("{0}. {1}".format(Count, Item))
    def ImprimirLista2(**kwargs):
        for Name, Value in kwargs.items():
            print("{0} pertenece a {1}".format(Name, Value))

MiClase.ImprimirLista1("Gryffindor", "Slytherin", "Ravenclaw")
MiClase.ImprimirLista2(Harry="Gryffindor", Draco="Slytherin", Luna="Ravenclaw")
```

Herencia (Clases Padre e Hijos)

En ocasiones vamos a necesitar generalizar Clases para poder reutilizarlas en clases hijas y de esta manera simplificar y ahorrar código. En el ejemplo anterior de los perros usábamos una clase Perro, sin embargo pueden haber muchas razas de Perro.

Para que una clase hija herede los atributos y métodos del padre simplemente la pasamos como parámetro en la definición de la clase.

```
class DogoArgentino(Perro):  
    pass  
  
class BulldogFrances(Perro):  
    pass  
  
class Labrador(Perro):  
    pass
```

```
>>> pancho = DogoArgentino("Pancho", 4)  
>>> roque = BulldogFrances("Roque", 9)  
>>> ramon = Labrador("Ramón", 3)  
>>> firulais = Labrador("Firulais", 5)
```

```
>>> pancho.especie  
>>> pancho.nombre  
>>> ramon.ladtrar
```

¿Cómo identifico las instancias?

Muchas veces necesitamos saber si un objeto es instancia de una clase en particular. Y lo hacemos mediante `isinstance(objeto, clase)` y nos devolverá `True` o `False` según corresponda:

```
>>> isinstance(pancho, Perro)
True
>>> isinstance(roque, Perro)
True
>>> isinstance(pancho, DogoArgentino)
True
>>> isinstance(pancho, BulldogFrances)
False
```

En el ejemplo anterior todos los objetos son instancias de `Perro` pero cada uno de ellos pertenece a una raza en particular.

Ampliando funcionalidades

En el ejemplo de los perros, el método ladrar no es el mismo para cada Perro, es decir cada uno ladra de diferentes maneras según su raza. Para determinar cada uno de los ladridos según la raza sobreescribimos el método ladrar cuando creamos la clase según la raza.

```
class DogoArgentino(Perro):  
    def ladrar(self, sonido="Arf"):  
        return (f"{self.nombre} dice {sonido}")  
  
class BulldogFrances(Perro):  
    def ladrar(self, sonido="Woof"):  
        return (f"{self.nombre} dice {sonido}")
```

Ademas puedes cambiar el valor del sonido durante la instancia simplemente llamando al método:

```
>>> ramon = DogoArgentino("Ramon", 5)  
>>> ramon.ladrar("Grrrrr")
```

Ampliando funcionalidades

En el ejemplo de los perros, el método ladrar no es el mismo para cada Perro, es decir cada uno ladra de diferentes maneras según su raza. Para determinar cada uno de los ladridos según la raza sobreescribimos el método ladrar cuando creamos la clase según la raza.

```
class DogoArgentino(Perro):  
    def ladrar(self, sonido="Arf"):  
        return (f"{self.nombre} dice {sonido}")  
  
class BulldogFrances(Perro):  
    def ladrar(self, sonido="Woof"):  
        return (f"{self.nombre} dice {sonido}")
```

Ademas puedes cambiar el valor del sonido durante la instancia simplemente llamando al método:

```
>>> ramon = DogoArgentino("Ramon", 5)  
>>> ramon.ladrar("Grrrrr")
```

Facilitando la herencia



En algunos casos no es necesario que sobreescribamos todo el método de la clase padre, por ejemplo si necesitamos solamente cambiar el parámetro del sonido del ladrido podemos hacer que la subclase llame al método de la clase padre mediante `super()`

```
class DogoArgentino(Perro):  
    def ladrar(self, sonido="Arf"):  
        return super().ladrar(sonido)
```

Cuando llamamos a `super().ladrar(sonido)` dentro de `DogoArgentino` Python busca un método llamado `ladrar` dentro de la clase padre `Perro` y le pasa el atributo `sonido`.



Orden de Resolución de Métodos



Probablemente se habrán imaginado a esta altura que puede existir herencia multiple (puedes crear una clase que herede de múltiples padres). Para ello Python utiliza un sistema denominado MRO o Multiple Resolution Order que se encarga de organizar el orden en el que los métodos pueden heredar.

Puedes ver la MRO usando el atributo `__mro__`

Si definiéramos nuevamente nuestra clase perro de ésta manera `Perro(AnimalTerrestre, Vertebrado)`, y `AnimalTerrestre(Mamifero)`, y `Mamifero(Animal)` y luego `Animal()` y quisiéramos averiguar el orden de MRO lo hacemos mediante:

```
>>> Perro.__mro__
(<class 'Perro'>,
 <class 'AnimalTerrestre'>,
 <class 'Vertebrado'>,
 <class 'Mamifero'>,
 <class 'Animal'>,
 <class 'object'>)
```

Operadores de Sobrecarga

```
class MiClase:
    def __init__(self, *args):
        self.Entrada = args
    def __add__(self, other):
        Salida = MiClase()
        Salida.Entrada = self.Entrada + other.Entrada
        return Salida
    def __str__(self):
        Salida = ""
        for Item in self.Entrada:
            Salida += Item
            Salida += " "
        return Salida

instancia1 = MiClase("Gryffindor", "Slytherin", "Ravenclaw")
instancia2 = MiClase("Harry", "Draco", "Luna")
instancia3 = instancia1 + instancia2
print("{0} + {1} = {2}".format(instancia1, instancia2, instancia3))
```

Otro ejemplo...

- ❖ Ahora, veamos un ejemplo más interesante en el que en lugar de almacenar solo las coordenadas de un punto, creamos una clase que representa el vuelo de una aerolínea:

```
class Vuelo():  
    # Metodo para crear un nuevo vuelo con una capacidad dada  
    def __init__(self, capacidad):  
        self.capacidad = capacidad  
        self.pasajeros = []  
  
    # Metodo para agregar un pasajero al vuelo:  
    def agregar_pasajero(self, nombre):  
        self.pasajeros.append(nombre)
```

Agreguemos más lógica

Sin embargo, esta clase tiene fallas porque aunque establezcamos una capacidad, aún podríamos agregar demasiados pasajeros. Modifiquemos para que antes de agregar un pasajero, verifiquemos si hay espacio en el vuelo:

```
class Vuelo():
    # Metodo para crear un nuevo vuelo con una capacidad dada
    def __init__(self, capacidad):
        self.capacidad = capacidad
        self.pasajeros = []

    # Metodo para agregar un pasajero al vuelo:
    def agregar_pasajero(self, nombre):
        if not self.asientos_disponibles():
            return False
        self.pasajeros.append(nombre)
        return True

    # Metodo para retornar el numero de asientos disponibles
    def asientos_disponibles(self):
        return self.capacidad - len(self.pasajeros)
```

Esto funciona porque en Python, el número 0 se puede interpretar como False, y también podemos usar la palabra clave not para significar lo opuesto a la siguiente declaración, por lo que `not True` es False y `not False` es True. Por lo tanto, si `asientos_disponibles` devuelve 0, toda la expresión se evaluará como True

Instanciando Vuelos

Ahora, probemos la clase que hemos creado creando instancias de algunos objetos:

```
# Crear un nuevo vuelo con hasta 3 personas
unVuelo = Vuelo(3)
# Crear una lista de personas
personas = ["Harry", "Ron", "Hermione", "Ginny"]
# Intentar agregar cada persona de la lista al vuelo
for unaPersona in personas:
    if unVuelo.agregar_pasajero(unaPersona):
        print(f"Agregado {unaPersona} al vuelo satisfactoriamente")
    else: print(f"No hay asientos disponibles para {unaPersona}")
```




Conclusión

Ésta ha sido una clase intensa y hemos aprendido que:

- En Python puedo adelantarme a los errores que puedan suceder y capturarlos mediante try
- Puedo representar los objetos del mundo real mediante clases en Python usando su capacidad de programación orientada a objetos.
- Lo que has aprendido aquí no solo te servirá para Python sino también para otros lenguajes que apliquen el paradigma de POO como Java, C# y C++.



Trabajemos Juntos



Llegó la hora del desafío:

1. Escribe una clase de Python llamada Rectangulo que se define por una longitud y un ancho y un método que calculará el área de un rectángulo.
2. Crea una clase Minibus que herede de la clase Vehiculo. Debes poder tener un método capacidad() que defina por defecto la capacidad de 6 asientos.

Luego crea una clase Pasajero que puedas ir agregando a una instancia de Minibus. Esa instancia no deberá permitir mas de 50 pasajeros únicos (no se permiten pasajeros repetidos).



#

¡HASTA LA
próxima!

www.polotic.misiones.gob.ar

[f](#) [@](#) [v](#) [d](#) /poloticmisiones

polotic
misiones



Gobierno
de Misiones

