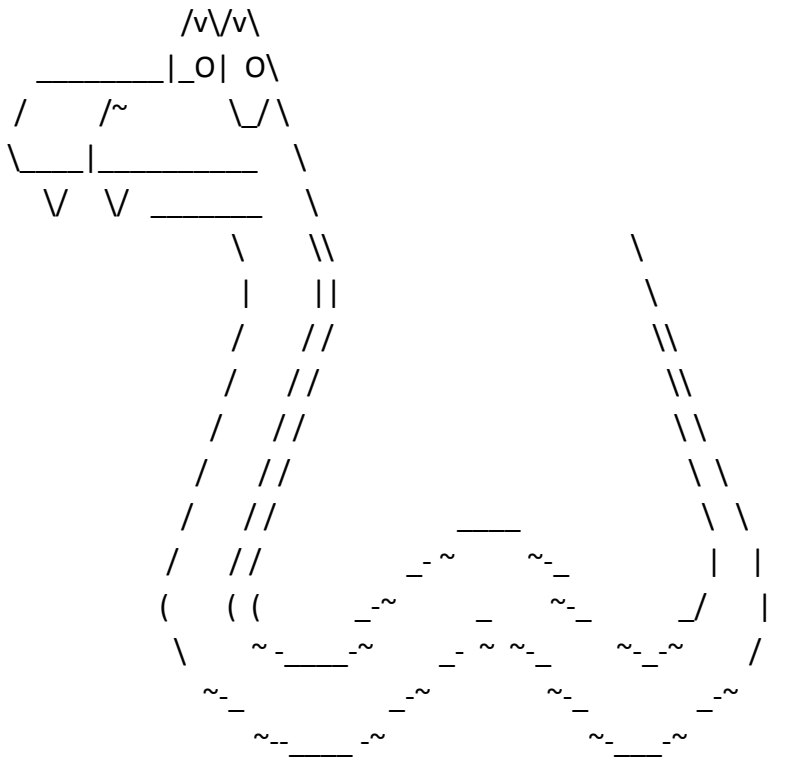
Python avanzado



El objetivo de este archivo es recopilar información de distintas bibliografías con el fin de formar un resumen el cual recopila las funciones y aplicaciones más avanzadas y complejas con las que cuenta Python.

Cualquiera que desee aprender sobre Python será capaz de descargar este archivo. De igual manera la bibliografía utilizada para este este resumen estará destacada por capítulo y se recomienda leerla a medida que se desee. El objetivo es que este resumen cumpla con el mismo fin de esta.

Parte I: Python Object Oriented Programming (OOP)

Object-Oriented Programming (OOP) es muy útil porque nos permite escribir un bloque de código y reutilizarlo creando diferentes clases e instancias de esas clases. Cada una de estas clases puede agregar nuevas funcionalidades sin necesidad de reescribir, cambiar o modificar el código existente. Los programas complejos requieren el uso de clases y objetos porque podemos dividir el código en múltiples partes manejables, lo que facilita su organización, mantenimiento y expansión a lo largo del tiempo.

Los principios de la Programación Orientada a Objetos

Los siguientes principios son generalmente aplicables a todos los lenguajes de programación orientados a objetos:

- **Abstracción**: Permite representar ideas complejas de forma simple mediante clases, ocultando los detalles internos y mostrando solo lo esencial.
- **Encapsulamiento**: Consiste en agrupar datos y métodos que operan sobre ellos en una sola unidad (la clase) y restringir el acceso directo a algunos de los componentes.
- **Herencia**: Permite que una clase hereda atributos y métodos de otra, fomentando la reutilización del código.
- **Polimorfismo**: Permite que objetos de distintas clases respondan al mismo método de diferentes formas.

¿Qué otros lenguajes soportan la Programación Orientada a Objetos?

El lenguaje orientado a objetos más antiguo y conocido es Java. Luego está C#, otro lenguaje OOP desarrollado por Microsoft. Algunos de los lenguajes OOP más famosos son PHP, Ruby, TypeScript y Python.

¿Cómo podemos crear clases en Python?

Para crear una clase en Python, usamos la palabra clave class, seguida del nombre de la clase. Por ejemplo:

```
class MiClase:
    pass
```

Esta estructura nos permite definir atributos y métodos que darán forma al comportamiento y características de los objetos que creemos a partir de ella.

Como puedes ver, el nombre de la clase comienza con letra mayúscula, y esta es una convención de nombres que deberías adoptar. Otra regla importante es que el nombre de la clase debe ser singular.

Si el nombre de la clase necesita estar compuesto por dos palabras, debes utilizar la convención PascalCase, donde cada palabra empieza con mayúscula y no se utilizan guiones bajos. Por ejemplo:

```
class UnEstudiante:
    pass
```

Esto mejora la legibilidad del código y sigue las buenas prácticas de programación en Python.

¿Qué son las Clases y los Objetos en Python?

En Python, las clases son como un plano o plantilla que usamos para crear objetos. Dentro de una clase, definimos una serie de propiedades (atributos) y métodos (funciones) que van a representar las características y comportamientos comunes de todos los objetos que creemos a partir de esa clase.

Por otro lado, los objetos (también llamados instancias) son ejemplares concretos creados a partir de esa clase. Cada objeto tiene su propia copia de las propiedades y puede usar los métodos definidos en la clase.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
    def saludar(self):
        print(f"Hola, mi nombre es {self.nombre} y tengo {self.edad} años.")
persona1 = Persona("Ana", 30) # Crear un objeto (instancia) de la clase Persona
persona1.saludar() # Usar un método
```

- **Clase (Class)**: Es la estructura que define qué atributos y métodos va a tener el objeto.
- **Objeto (Object/Instance):** Es una instancia real creada a partir de una clase.
- **Propiedad (Property):** Es una variable que representa una característica del objeto (por ejemplo, nombre, color, edad).
- **Método (Method):** Es una función definida dentro de una clase que define acciones o comportamientos que el objeto puede realizar.

Construir clases, atributos y métodos

En Python, los métodos de clase que comienzan y terminan con doble guion bajo (por ejemplo, __init__) se conocen como métodos especiales, dunder methods (por "double underscore") o métodos mágicos.

Estos métodos tienen un propósito especial dentro de las clases. Uno de los más comunes es __init__, también llamado constructor, que se ejecuta automáticamente cuando se crea un nuevo objeto de esa clase.

Aunque los métodos usan la misma sintaxis que las funciones (es decir, comienzan con la palabra clave def), la diferencia clave es que:

- Las funciones existen en el ámbito global (fuera de una clase).
- Los métodos están definidos dentro de una clase y son usados por los objetos que se crean a partir de esa clase.

Por eso, en la documentación o libros, muchas veces se habla de funciones dentro de clases como si fueran intercambiables, pero técnicamente, si están dentro de una clase, se consideran métodos.

- def: se usa tanto para definir funciones como métodos.
- __init__: es un método especial llamado automáticamente al crear un objeto.
- __nombre__: cualquier método con doble guion bajo es un método especial (como __str__, __len__, etc.).

Un atributo de objeto de clase (o atributo de clase) es distinto de los demás atributos definidos dentro del método __init__ porque no pertenece a una instancia específica, sino a la clase en sí. Es decir, es compartido por todos los objetos creados a partir de esa clase.

```
class Persona:
    is_person = True  # Atributo de clase

    def __init__(self, nombre, apellido, edad):
        self.nombre = nombre
        self.apellido = apellido
        self.edad = edad

    def saludar(self):
        print(f"Hola, me llamo {self.nombre} {self.apellido} y tengo {self.edad} años.")
```

En este caso:

- is_person es un atributo de clase: se define fuera de __init__ y no depende de un objeto particular.
- Se puede acceder a él desde cualquier instancia (por ejemplo, persona1.is_person) o directamente desde la clase (Persona.is_person).
- Los atributos como nombre, apellido y edad son atributos de instancia, porque se asignan en __init__ con self.

```
persona1 = Persona("Ana", "López", 28)
print(persona1.is_person)     # True
```

La palabra clave self se usa en los métodos para referirse a la instancia que invoca el método, y así acceder a sus atributos individuales. En cambio, los atributos estáticos como is_person no necesitan self para ser llamados desde la clase.

Los atributos de clase en Python son variables que se definen directamente dentro de la clase y fuera de cualquier método, por lo que son compartidos por todas las instancias u objetos que se creen a partir de esa clase. Se consideran estáticos, ya que su valor no cambia entre objetos, a menos que se modifique desde la clase misma. En cambio, los atributos de instancia se definen dentro del método __init__ usando self, y cada objeto creado tendrá su propia copia de estos atributos con valores individuales. Esto permite que una clase contenga tanto información general (común a todos los objetos) como información específica de cada instancia.

El método __init__()

El método __init__ en Python es conocido como el constructor de una clase y se ejecuta automáticamente cada vez que se crea una nueva instancia de dicha clase. Su función principal es inicializar los atributos del objeto, es decir, asignar los valores iniciales que definen su estado. Se utiliza la palabra clave self para referirse a la propia instancia que se está creando. Aunque ya se explicó su uso básico, es importante destacar que el __init__ puede aceptar tantos parámetros como se necesiten, lo que permite personalizar cada objeto desde su creación. Además, se pueden incluir dentro del constructor validaciones, condiciones o incluso llamadas a otros métodos si se requiere una configuración más compleja del objeto al momento de su instalación.

El constructor __init__ cumple con la función de crear una nueva instancia u objeto a partir de una clase, como por ejemplo la clase Person. Este nuevo objeto podrá usar los atributos y métodos definidos en la clase. El primer parámetro del constructor es siempre la palabra clave self, que actúa como una referencia al propio objeto que se está creando. Aunque self parece un parámetro más, es especial porque permite acceder y diferenciar los datos de cada

instancia. Cada objeto que creamos con la clase tendrá su propio self, lo que garantiza que cada instancia sea única y maneje su propia información de manera independiente. Al principio puede parecer confuso, pero self es esencial para entender cómo funcionan los objetos en Python.

Crear métodos de clase utilizando @classmethod

Para crear métodos de clase en Python que pertenezcan directamente a la clase y no a una instancia específica, usamos el decorador @classmethod. Este decorador se coloca justo antes de la definición del método, y dentro del método usamos el parámetro cls (abreviatura de class) en lugar de self. El parámetro cls hace referencia a la clase y permite acceder o modificar atributos de clase.

```
class Persona:
    especie = "Humano"

    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    @classmethod
    def cambiar_especie(cls, nueva_especie):
        cls.especie = nueva_especie
```

---

cambiar_especie es un método de clase que permite modificar el atributo especie para todas las instancias de la clase Persona, porque especie es un atributo estático (de clase). Al usar @classmethod, este método puede ser llamado así:

```
Persona.cambiar_especie("Alienígena")
```

Para resumir lo que hemos aprendido sobre los métodos de clase en Python:

- El método de clase está vinculado a la clase y no a una instancia (objeto) específica.
- Utiliza el parámetro cls, que representa la clase misma, y no self, que representa una instancia.
- Tiene acceso a todos los atributos de clase (también llamados atributos estáticos), no a los atributos individuales de cada objeto.
- Puede acceder y modificar el estado de la clase, lo cual es útil para mantener un valor compartido entre todas las instancias.

Metodo estatico @staticmethod

El método estático en Python se define usando el decorador @staticmethod y es una función que pertenece a la clase, pero no necesita acceso ni a la instancia (self) ni a la clase (cls). Es decir:

- No accede ni modifica atributos de instancia ni de clase.
- Es útil para crear métodos utilitarios que están relacionados conceptualmente con la clase, pero que no dependen de su estado interno.
- Se puede llamar tanto desde la clase como desde una instancia, pero su comportamiento será siempre el mismo.

```
class Calculadora:

    @staticmethod
    def sumar(a, b):
        return a + b

# Llamada desde la clase
print(Calculadora.sumar(5, 3))

# Llamada desde una instancia
calc = Calculadora()
print(calc.sumar(10, 7))
```

Esto es especialmente útil para mantener el código organizado: puedes agrupar funciones relacionadas dentro de una clase sin que estas funciones necesiten acceder al estado de dicha clase.

1er pilar de la programación orientada a objetos - Encapsulado

Encapsulation es un principio fundamental de la programación orientada a objetos (OOP) que consiste en agrupar en una misma clase tanto los datos (atributos) como los métodos (funciones) que manipulan esos datos. Esto crea una especie de "caja" o "paquete" que protege el estado interno del objeto, permitiendo el acceso y la modificación sólo a través de métodos específicos. Por ejemplo, en una clase Persona, los atributos como nombre y edad están encapsulados junto con métodos como saludar() o cumplir_años(). Esta estructura no solo organiza el código, sino que también mejora la seguridad y el control, ya que se puede restringir el acceso directo a ciertos atributos usando modificadores como _atributo o __atributo para indicar distintos niveles de privacidad.

```
class Persona:
    def __init__(self, nombre, edad):
        self.__nombre = nombre  # Atributo privado
        self.__edad = edad      # Atributo privado

    def saludar(self):
        print(f"Hola, mi nombre es {self.__nombre} y tengo {self.__edad} años.")

    def cumplir_años(self):
        self.__edad += 1

    def obtener_edad(self):
        return self.__edad

    def establecer_nombre(self, nuevo_nombre):
        self.__nombre = nuevo_nombre


persona1 = Persona("Ana", 30)
persona1.saludar()  # Hola, mi nombre es Ana y tengo 30 años.
persona1.cumplir_años()
print(persona1.obtener_edad())  # 31
persona1.establecer_nombre("Ana María")
persona1.saludar()  # Hola, mi nombre es Ana María y tengo 31 años.
```

¿Qué muestra la encapsulación aquí?

- Los atributos __nombre y __edad están encapsulados, es decir, no son accesibles directamente desde fuera de la clase.
- Solo se pueden modificar o leer usando métodos definidos, lo que proporciona control y seguridad sobre cómo se accede a los datos internos.

2do pilar de la programación orientada a objetos - Abstracción

Abstracción significa ocultar los detalles internos de implementación y mostrar solo lo esencial para el usuario. Esto ayuda a simplificar la interacción con objetos complejos, mostrando únicamente lo que se necesita.

```
class Persona:
    def __init__(self, nombre, edad):
        self.__nombre = nombre
        self.__edad = edad

    def __procesar_nombre(self):
        # Método interno que no necesita conocer el usuario
        return self.__nombre.strip().title()

    def saludar(self):
        nombre_limpio = self.__procesar_nombre()
        print(f"Hola, mi nombre es {nombre_limpio} y tengo {self.__edad} años.")

persona1 = Persona("   ana", 25)
persona1.saludar()  # Hola, mi nombre es Ana y tengo 25 años.
```

¿Dónde está la abstracción?

- El método __procesar_nombre() es interno (privado). El usuario de la clase no necesita saber cómo se procesa el nombre, solo le interesa que saludar() funcione correctamente.
- El usuario accede a una interfaz simple (saludar()) sin preocuparse por la lógica interna.

Variables publicas vs privadas

En Python, las variables privadas y públicas se usan para controlar el acceso a los atributos de una clase. Aunque Python no tiene encapsulación estricta como otros lenguajes (por ejemplo, Java o C++), sigue ciertas convenciones que ayudan a marcar una variable como privada o pública.

Variables publicas

Estas variables se pueden acceder y modificar libremente desde fuera de la clase.

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre  # pública

p = Persona("Ana")
print(p.nombre)  # Acceso permitido
p.nombre = "Laura"  # Modificación permitida
print(p.nombre)
```

Variables Privadas

En Python, las variables privadas se indican anteponiendo doble guión bajo (__) al nombre de la variable. Esto activa el name mangling, haciendo que no se pueda acceder directamente desde fuera de la clase.

```
class Persona:
    def __init__(self, nombre):
        self.__nombre = nombre  # privada

p = Persona("Ana")
# print(p.__nombre) # Esto da error: AttributeError

# Acceso usando name mangling (no recomendado)
print(p._Persona__nombre)  #  Acceso posible pero no es una buena práctica
```

- Para proteger datos sensibles.
- Para evitar modificaciones accidentales desde fuera de la clase.
- Para forzar a usar métodos de acceso como getters y setters.

3er pilar de la programación orientada a objetos - Herencia

Es uno de los pilares fundamentales de la programación orientada a objetos. Permite crear una nueva clase (llamada clase hija o derivada) que hereda los atributos y métodos de una clase existente (llamada clase padre o base), lo que promueve la reutilización del código y la organización jerárquica. En Python, esto se logra indicando entre paréntesis el nombre de la clase padre al definir la clase hija. La clase hija puede extender la funcionalidad de la clase padre agregando nuevos atributos o métodos, o incluso sobrescribiendo los existentes. A continuación, se muestra un ejemplo donde la clase Student hereda de la clase Person, aprovechando el constructor y el método greet definidos en la clase base:

```
# Clase base (padre)
class Person:
    def __init__(self, name, lastname, age):
        self.name = name
        self.lastname = lastname
        self.age = age

    def greet(self):
        print(f"Hola, soy {self.name} {self.lastname} y tengo {self.age} años.")

# Clase derivada (hija)
class Student(Person):
    def __init__(self, name, lastname, age, student_id):
        super().__init__(name, lastname, age)  # Llamamos al constructor de la clase padre
        self.student_id = student_id

    def mostrar_estudiante(self):
        print(f"Soy un estudiante con ID: {self.student_id}")

# Creamos una instancia de Student
estudiante = Student("Lucía", "González", 21, "A12345")
estudiante.greet()              # Método heredado de la clase Person
estudiante.mostrar_estudiante()  # Método propio de Student
```

- super().__init__() permite llamar al constructor de la clase padre (Person) para reutilizar código.
- Student hereda automáticamente todos los métodos y atributos públicos de Person.
- Podés añadir más métodos o atributos en la clase hija, o incluso sobrescribir los del padre si necesitás comportamiento diferente.

Sobrescribir métodos

El método overriding (sobrescritura de métodos) ocurre cuando una clase hija redefine un método heredado de su clase padre, utilizando el mismo nombre pero con una implementación distinta. Esto es útil cuando queremos que la clase hija personalice o amplíe el comportamiento del método heredado. En el siguiente ejemplo, la clase Person tiene un método greet() que muestra el nombre y apellido.

```
# Clase base (padre)
class Person:
    def __init__(self, name, lastname, age):
        self.name = name
        self.lastname = lastname
        self.age = age

    def greet(self):
        print(f"Hola, soy {self.name} {self.lastname} y tengo {self.age} años.")

# Clase hija con método sobrescrito
class Student(Person):
    def __init__(self, name, lastname, student_id):
        super().__init__(name, lastname)
        self.student_id = student_id

    def greet(self):  # Sobrescribimos el método greet
        print(f"Soy {self.name} {self.lastname}, mi ID de estudiante es {self.student_id}.")

# Creamos una instancia de Student
persona = Person("Carlos", "Pérez")
estudiante = Student("Ana", "López", "S1234")

persona.greet()     # Imprime: Hola, soy Carlos Pérez.
estudiante.greet()  # Imprime: Hola, soy Ana López y mi ID de estudiante es S1234.
```

Función isinstance()

La función isinstance() en Python se utiliza para verificar si un objeto pertenece a una clase específica o a una subclase de esa clase. Esto es especialmente útil cuando trabajamos con herencia, ya que nos permite comprobar si un objeto es instancia de una clase padre, una clase hija o ambas. Recordando lo aprendido, las clases en Python nos permiten definir estructuras reutilizables, y gracias a la herencia, una clase hija puede heredar atributos y métodos de una clase padre.

```
# Clase padre
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

# Clase hija
class Perro(Animal):
    def __init__(self, nombre, raza):
        super().__init__(nombre)
        self.raza = raza

# Creamos una instancia de Perro
mi_mascota = Perro("Firulais", "Labrador")

# Verificamos con isinstance()
print(isinstance(mi_mascota, Perro))   # True
print(isinstance(mi_mascota, Animal))  # True
print(isinstance(mi_mascota, object))  # True
```

El número de atributos y métodos de una clase depende de lo que represente, por lo que puede tener muchos o ninguno. Uno de sus métodos más importantes es el constructor __init__, donde el primer parámetro es self (referencia al objeto que se está creando) y los siguientes son los atributos que queremos inicializar. La creación de objetos a partir de una clase se conoce como instanciación, y cada objeto es una instancia de esa clase. Al instanciar, el constructor se ejecuta automáticamente y, gracias al parámetro self, la clase sabe a qué objeto asignarle los valores. Los métodos dentro de una clase funcionan como funciones, pero se les llama métodos porque pertenecen a una clase y pueden ser utilizados tanto por la propia clase como por sus objetos. Además, existen atributos y métodos de clase que pueden usarse sin necesidad de instanciar la clase. Finalmente, la herencia permite que una clase hija reutilice los atributos y métodos de una clase padre, lo que evita la duplicación de código y mejora la eficiencia.

4to pilar de la programación orientada a objetos - Polimorfismo

El cuarto pilar de la Programación Orientada a Objetos (OOP) es el Polimorfismo, una palabra derivada del griego que significa “muchas formas”. Este concepto permite que una misma función o método tenga el mismo nombre pero se comporte de manera diferente según el contexto o la clase en la que se utilice. Es decir, diferentes clases pueden definir un método con el mismo nombre pero implementarlo de forma distinta. Por ejemplo, en una relación de herencia entre una clase padre y una clase hija, ambas pueden tener un método llamado eat, pero mientras en la clase padre el método representa una acción general, en la clase hija puede adaptarse a un comportamiento más específico. Esta capacidad de modificar o extender comportamientos sin cambiar el nombre del método es clave para la reutilización del código y la flexibilidad en los programas orientados a objetos.

```
# Objeto tipo cadena
str_obj = "Polimorfismo"

# Objeto tipo lista
list_obj = ["Java", "Python", "Ruby", "C#", "PHP", "C++"]

# Objeto tipo diccionario
dict_obj = {
   "marca": "Tesla",
   "modelo": "Nombres divertidos :)",
   "año": 2023
}

# La función len() se comporta diferente según el tipo de objeto
print(len(str_obj))    # Devuelve la cantidad de caracteres de la cadena
print(len(list_obj))   # Devuelve la cantidad de elementos en la lista
print(len(dict_obj))   # Devuelve la cantidad de claves en el diccionario
```

Este es un gran ejemplo de polimorfismo, ya que la función len() se aplica a diferentes tipos de objetos (string, lista y diccionario), pero se adapta y devuelve un resultado acorde al tipo de dato.

super()

La función super() se utiliza en programación orientada a objetos (OOP) para llamar métodos o atributos de una clase padre desde una clase hija. El programa debe tener dos clases. La primera clase se llamará Personal y la segunda clase se llamará Employee. Tenés que deducir cuál es la relación entre ambas para poder escribir correctamente la herencia.

La clase Person debe tener los siguientes atributos:

- name (nombre)
- dob (fecha de nacimiento)
- id_number (número de identificación)

Dentro de la clase Person, debés crear un método llamado info() que imprima todos los detalles de la persona, cada uno en una nueva línea.

La clase Employee debe poder utilizar todos los atributos y métodos de la clase padre, pero también debe tener nuevos atributos:

- salary (salario)
- position (puesto)

Esta clase Employee también tendrá su propio método info() con el mismo nombre que el método de la clase Person, pero modificado para que imprima tanto los detalles heredados de la clase Person como los nuevos detalles del empleado. Este proceso se llama sobrescritura de métodos (method overriding).

```
# Clase padre
class Person:
    def __init__(self, name, dob, id_number):
        self.name = name
        self.dob = dob
        self.id_number = id_number

    def info(self):
        print("Nombre:", self.name)
        print("Fecha de nacimiento:", self.dob)
        print("Número de ID:", self.id_number)

# Clase hija
class Employee(Person):
    def __init__(self, name, dob, id_number, salary, position):
        # Usamos super() para llamar al constructor de la clase padre
        super().__init__(name, dob, id_number)
```

```
        self.salary = salary
        self.position = position

    def info(self):
        # Llamamos al método info() de la clase padre
        super().info()
        print("Salario:", self.salary)
        print("Puesto:", self.position)

# Crear una instancia de la clase Person
persona = Person("Ana López", "1990-04-15", "12345678")
print("----- Información de la persona -----")
persona.info()

# Crear una instancia de la clase Employee
empleado = Employee("Carlos Méndez", "1985-11-23", "87654321", 50000, "Gerente")
print("\n----- Información del empleado -----")
empleado.info()
```

Introspección de código en Python

La introspección es la capacidad que tiene Python para inspeccionar objetos en tiempo de ejecución. Esto significa que podemos averiguar qué tipo de objeto es, cuáles son sus atributos, métodos disponibles, y de qué clase proviene, mientras el programa se está ejecutando. Esto es posible porque en Python todo es un objeto, y todos los objetos se heredan de la clase base object.

- type(): Devuelve el tipo de objeto.
- id(): Devuelve el identificador único del objeto en memoria.
- dir(): Muestra todos los atributos y métodos disponibles para un objeto.
- hasattr(obj, 'attr'): Verifica si un objeto tiene un atributo específico.
- getattr(obj, 'attr'): Devuelve el valor de un atributo si existe.
- callable(): Verifica si un objeto puede ser llamado (por ejemplo, si es una función).
- isinstance(obj, Clase): Comprueba si un objeto es una instancia de cierta clase.

```
print(type(5))          # <class 'int'>
print(type("Hola"))     # <class 'str'>

x = 42
print(id(x))  # Ej: 140737488351312

print(dir("Python"))
```

```
class Persona:
    nombre = "Juan"

p = Persona()
print(hasattr(p, 'nombre'))  # True

print(getattr(p, 'nombre'))  # Juan

def saludar():
    print("Hola")
print(callable(saludar))  # True

print(isinstance(p, Persona))  # True
```

Dunder o Magic Methods

En Python, los Dunder Methods (abreviación de Double UNDerscore Methods) son funciones especiales que comienzan y terminan con dos guiones bajos, como __init__, __str__, __len__, entre muchos otros. Se les llama también métodos mágicos porque Python los ejecuta automáticamente en ciertos contextos sin que el programador tenga que llamarlos explícitamente.

Por ejemplo, cuando creamos un nuevo objeto de una clase, Python llama automáticamente al método __init__, que es el constructor de la clase. Del mismo modo, si usamos la función print() con un objeto, Python invoca internamente el método __str__, que devuelve una representación legible del objeto. Así, estos métodos nos permiten personalizar cómo se comportan nuestros objetos frente a ciertas funciones o símbolos del lenguaje.

Un aspecto interesante de los dunder methods es que nos permiten sobrecargar operadores. Por ejemplo, podemos usar el método __add__ para definir qué debe hacer el símbolo + entre dos objetos de nuestra clase. También existe __len__ para que nuestros objetos puedan usarse con la función len(), y __getitem__ para acceder a los elementos de un objeto como si fueran una lista (obj[0], por ejemplo).

Este sistema permite que nuestras clases se integren de forma muy natural con el lenguaje, logrando objetos más intuitivos, flexibles y potentes. Además, con el uso de introspección como dir(objeto), podemos ver todos estos métodos disponibles y entender mejor cómo funcionan internamente.

Lista de métodos más comunes

- __init__: Constructor, inicializa objetos
- __str__: Representación en string (print)
- __repr__: Representación para depuración
- __len__: Longitud del objeto (len)
- __getitem__: Acceder con índice (obj[i])
- __setitem__: Asignar con índice (obj[i] = valor)
- __add__: Suma con +
- __eq__: Comparar con ==

```
class Libro:
    def __init__(self, titulo, paginas):
        self.titulo = titulo
        self.paginas = paginas

    def __str__(self):
        return f"Libro: {self.titulo} ({self.paginas} páginas)"

    def __len__(self):
        return self.paginas

    def __add__(self, otro_libro):
        return self.paginas + otro_libro.paginas

# Crear objetos
libro1 = Libro("Python Fácil", 300)
libro2 = Libro("POO en Profundidad", 200)

print(libro1)            # __str__: Libro: Python Fácil (300 páginas)
print(len(libro1))       # __len__: 300
print(libro1 + libro2)   # __add__: 500
```

Herencia Múltiple en Python

La herencia en programación orientada a objetos permite que una clase (llamada clase hija o derivada) hereda atributos y métodos de otra clase (llamada clase padre o base). Sin embargo, la herencia no se limita a un solo tipo. Existen distintos tipos de herencia, como:

- Herencia simple
- Herencia multinivel
- Herencia múltiple
- Herencia multipath
- Herencia híbrida
- Herencia jerárquica

En este caso, nos vamos a enfocar en la herencia múltiple. Este tipo de herencia ocurre cuando una clase hija hereda de dos o más clases padre al mismo tiempo. A diferencia de la herencia simple, donde solo hay una línea de herencia, la herencia múltiple permite combinar funcionalidades de distintas clases en una sola.

Por ejemplo, imagina que tenemos una clase Estudiante y una clase Atleta. Si queremos crear una clase EstudianteDeportivo, podríamos hacer que herede tanto de Estudiante como de Atleta. De esta manera, EstudianteDeportivo podrá usar los atributos y métodos de ambas clases padre.

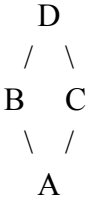
Python soporta herencia múltiple de forma directa. Solo tenemos que especificar los nombres de las clases padre entre paréntesis, separados por comas, al definir la clase hija. Sin embargo, con esta libertad también viene una posible complicación: el orden de resolución de métodos (Method Resolution Order - MRO). En casos donde dos clases tengan métodos con el mismo nombre, Python sigue una jerarquía para decidir cuál usar primero.

```
# Clase 1
class Estudiante:
    def __init__(self, nombre, materia):
        self.nombre = nombre
        self.materia = materia

    def mostrar_estudiante(self):
        print(f"Estudiante: {self.nombre}, Materia: {self.materia}")

# Clase 2
class Atleta:
    def __init__(self, deporte):
        self.deporte = deporte
```

```
    def mostrar_deporte(self):
        print(f"Deporte: {self.deporte}")

# Clase hija que hereda de Estudiante y Atleta
class EstudianteDeportivo(Estudiante, Atleta):
    def __init__(self, nombre, materia, deporte):
        # Llamamos a los constructores de ambas clases padre
        Estudiante.__init__(self, nombre, materia)
        Atleta.__init__(self, deporte)

    def mostrar_todo(self):
        self.mostrar_estudiante()
        self.mostrar_deporte()

persona = EstudianteDeportivo("Lucía", "Matemática", "Natación")
persona.mostrar_todo()
```

MRO - Method resolution order (Orden de resolución de los métodos)

MRO es el orden en el que Python busca métodos o atributos en una jerarquía de clases. Esto es especialmente importante cuando trabajamos con herencia múltiple, ya que puede haber ambigüedad si varias clases definen el mismo método. Python resuelve esto usando un algoritmo llamado C3 Linearization, que básicamente define un orden lineal y consistente en el cual buscar métodos.



La **linealización C3** es un algoritmo que utiliza Python para determinar el orden en que se deben buscar las clases padre cuando se aplica herencia múltiple. Este orden, llamado MRO (Method Resolution Order), asegura que las clases se recorran de forma coherente y predecible al resolver métodos y atributos heredados.

El proceso consiste en sumar la clase actual a una combinación (merge) especial de las linealizaciones de sus clases padre y la lista de padres directa. Esta combinación se hace manteniendo el orden local (el orden en que se indican los padres) y evitando duplicaciones o inconsistencias.

Durante el merge, se elige el primer elemento de cualquier lista que no aparezca en la parte restante (tail) de ninguna de las demás listas. Ese elemento se considera "válido" y se agrega al resultado. Luego, se elimina de todas las listas donde aparece como cabeza. Este proceso se repite hasta que todas las listas están vacías. Si en algún momento no se puede seleccionar ningún elemento válido, significa que la jerarquía de herencia tiene un orden inconsistente y no se puede calcular la MRO.

```
class D:
    def mostrar(self):
        print("Método de D")

class B(D):
    def mostrar(self):
        print("Método de B")

class C(D):
    def mostrar(self):
        print("Método de C")

class A(B, C):
    pass

obj = A()
obj.mostrar()
```

Otro algoritmo es **Depth First Search (DFS)** el cual es un algoritmo clásico utilizado para recorrer o buscar nodos dentro de estructuras como árboles o grafos. Su lógica consiste en avanzar lo más profundo posible por un camino antes de retroceder y explorar otros caminos. DFS no considera ninguna regla especial sobre jerarquía o dependencia entre nodos más allá de la conexión entre ellos. Se usa comúnmente para tareas como detectar ciclos, encontrar componentes conexos o generar caminos en estructuras de datos.

Por otro lado, C3 Linearization es una técnica específica de los lenguajes de programación orientados a objetos (como Python) que implementa herencia múltiple. Su objetivo es definir un orden consistente para resolver métodos cuando una clase hereda de múltiples clases padre. A diferencia de DFS, la C3 Linearization tiene reglas estrictas para preservar el orden local de herencia (el orden en que se listan las clases padre) y evita conflictos en la jerarquía que podrían generar ambigüedades o comportamientos inesperados.

La gran diferencia es que DFS simplemente explora estructuras sin preocuparse por el significado jerárquico de los nodos, mientras que C3 Linearization fusiona cuidadosamente los lineamientos de herencia respetando prioridades y evitando que un mismo ancestro se repita en posiciones conflictivas. Además, C3 incluye mecanismos para detectar y prevenir ciclos, algo que DFS solo detecta pero no maneja con lógica de herencia.

```
def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()
    visited.add(node)
    print(node)  # acción al visitar el nodo
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

dfs(graph, 'A')
```

Parte II: Automatización

Coincidencia de Patrones con Expresiones Regulares

Las expresiones regulares (regex) te permiten especificar un patrón de texto para buscar. Por ejemplo, tal vez no sepas el número exacto de teléfono de un negocio, pero si vivís en Estados Unidos o Canadá, sabés que tendrá tres dígitos, seguido de un guión, y luego cuatro dígitos más (y opcionalmente, un código de área de tres dígitos al inicio). Así es como, como persona, sabés que estás viendo un número de teléfono 415-555-1234 es un número válido, pero 4,155,551,234 no lo es.

Las expresiones regulares son herramientas poderosas y útiles, aunque no muchos no-programadores las conocen. Sin embargo, la mayoría de los editores de texto modernos y procesadores de texto como Microsoft Word u OpenOffice tienen funciones de "buscar" y "buscar y reemplazar" que permiten utilizar expresiones regulares. Estas expresiones son grandes ahorradoras de tiempo, tanto para usuarios comunes como para programadores, ya que permiten automatizar tareas repetitivas y encontrar patrones complejos en grandes volúmenes de texto.

Encontrar patrones de texto sin expresiones regulares

Supongamos que querés buscar un número de teléfono dentro de un texto. Sabés que el patrón es algo como: tres números, un guión, tres números, otro guión, y luego cuatro números, por ejemplo 415-555-4242. Una forma de detectar este patrón sería escribir una función que lo revise carácter por carácter.

```
def isPhoneNumber(text):
    if len(text) != 12:
        return False
    for i in range(0, 3):
        if not text[i].isdecimal():
            return False
    if text[3] != '-':
        return False
    for i in range(4, 7):
        if not text[i].isdecimal():
            return False
    if text[7] != '-':
        return False
    for i in range(8, 12):
        if not text[i].isdecimal():
            return False
    return True
```

```
# Ejemplo de uso:
print(isPhoneNumber("415-555-4242"))  # True
print(isPhoneNumber("Hola-mundo-2025"))  # False
```

Si solo usás funciones como isPhoneNumber() para detectar patrones simples, vas a necesitar mucho más código si querés buscar ese patrón dentro de un texto más largo, no solo verificar si una cadena es un número de teléfono.

```
# Buscar dentro de un texto largo
message = "Call me at 415-555-1011 tomorrow. 123-456-7890 is my office number."

for i in range(len(message)):
    chunk = message[i:i+12]
    if isPhoneNumber(chunk):
        print("Phone number found:", chunk)
```

Esto va revisando bloque por bloque de 12 caracteres, y chequea si ese bloque es un número de teléfono. Se toma un bloque de 12 caracteres desde la posición i hasta i+12 (no inclusivo). En cada iteración, chunk contiene una subcadena del texto original, y esa subcadena se compara con el patrón de número telefónico.

Esto significa que:

- En la 1ra vuelta: i = 0 → chunk = message[0:12] = 'Call me at 4'
- En la 2da vuelta: i = 1 → chunk = message[1:13] = 'all me at 41'
- En la 3ra vuelta: i = 2 → chunk = message[2:14] = 'll me at 415'
- ...

Y así sigue, hasta que encuentra una subcadena como '415-555-1011' que cumple con las condiciones de isPhoneNumber().

Este método funciona, pero es ineficiente y limitado. Imagina si tenés que buscar varios patrones diferentes, o si los formatos cambian. Ahí es donde las expresiones regulares se vuelven útiles ya que te permiten definir ese patrón con una sola línea y buscarlo directamente en todo el texto.

Encontrar patrones de texto con expresiones regulares

El uso de expresiones regulares (regex) en Python permite encontrar patrones de texto de manera mucho más eficiente y flexible que utilizando condicionales y bucles tradicionales. Por ejemplo, en lugar de escribir una función extensa para validar un número de teléfono con un formato específico como 415-555-4242, se puede usar un patrón regex como \d{3}-\d{3}-\d{4}, que realiza la misma tarea en una sola línea.

Las expresiones regulares funcionan describiendo la estructura del texto que se quiere encontrar. En el caso de los números telefónicos, \d representa un dígito y {3} indica que ese dígito debe repetirse tres veces. Al combinar estos elementos con guiones, se puede representar fácilmente el formato esperado.

Las expresiones regulares permiten adaptarse a variaciones en los formatos, como números con puntos en lugar de guiones, paréntesis alrededor del código de área o extensiones agregadas al final. Todo esto se puede capturar modificando ligeramente el patrón original, lo cual sería muy difícil de mantener con una solución manual. Por estas razones, las regex son herramientas fundamentales tanto para usuarios como para programadores.

Para trabajar con expresiones regulares en Python, primero necesitas importar el módulo re, que contiene todas las funciones necesarias para crear y utilizar expresiones regulares. Este módulo te permite compilar patrones en objetos especiales llamados regex objects, que luego puedes usar para buscar coincidencias en cadenas de texto.

La función principal para crear un objeto regex es re.compile(). Esta función toma como argumento una cadena que representa el patrón regex y devuelve un objeto que puede ser reutilizado varias veces para hacer búsquedas, coincidencias o sustituciones en diferentes textos.

```
import re

phoneRegex = re.compile(r'\d{3}-\d{3}-\d{4}')
```

Coincidencias con un objeto regex

Cuando creas un objeto de expresión regular (regex) usando re.compile(), puedes usar su método .search() para buscar una coincidencia dentro de una cadena de texto. Este método examina la cadena completa en busca de la primera aparición del patrón definido.

Si encuentra una coincidencia, devuelve un objeto Match. Si no encuentra nada, devuelve None. Este comportamiento te permite fácilmente verificar si un patrón existe en un texto o no.

```
phoneRegex = re.compile(r'\d{3}-\d{3}-\d{4}')
mo = phoneRegex.search('My number is 415-555-4242.')

if mo:
   print('Phone number found:', mo.group())
else:
   print('No phone number found.')
```

En este caso, .search() encuentra el patrón en el texto y retorna un objeto Match. Luego puedes usar .group() para obtener la cadena que coincidió con la expresión regular. Si el patrón no aparece, la variable mo será None y puedes manejarlo con una simple condición if.

Agrupar con paréntesis

Cuando utilizas paréntesis en una expresión regular en Python, estás creando grupos que permiten extraer partes específicas del texto coincidente. Por ejemplo, si quieres separar el código de área de un número telefónico del resto del número, puedes usar una expresión (\d\d\d)-(\d\d\d-\d\d\d\d). Aquí, el primer grupo (\d\d\d) captura los primeros tres dígitos (el código de área), y el segundo grupo (\d\d\d-\d\d\d\d) captura el resto del número.

Una vez que tienes un objeto Match al usar re.search() o re.match(), puedes acceder a los grupos con el método .group():

- match.group(0) devuelve todo lo que coincidió.
- match.group(1) devuelve el contenido del primer grupo.
- match.group(2) devuelve el contenido del segundo grupo, y así sucesivamente.

```
text = "Llama al 415-555-4242"
pattern = re.compile(r"(\d{3})-(\d{3}-\d{4})")
match = pattern.search(text)

if match:
   print("Código de área:", match.group(1))      # 415
   print("Número:", match.group(2))              # 555-4242
```

Agrupar con carácter "pipe"

El carácter |, conocido como pipe, se utiliza en expresiones regulares para indicar una alternativa: permite que coincida uno entre varios patrones posibles. Es como decir "o" en lenguaje natural.

```
pattern = re.compile(r"gato|perro")
text = "Mi mascota es un perro."

match = pattern.search(text)

if match:
   print("Encontrado:", match.group())
```

Cuando usas el pipe, la búsqueda para en la primera coincidencia válida, así que si tienes opciones similares, el orden importa.

Coincidencia opcional con un signo de pregunta

En expresiones regulares, el carácter ? se utiliza para indicar que el elemento justo antes de él es opcional, es decir, puede aparecer una vez o no aparecer en absoluto. Esto es útil cuando quieres que un patrón coincida con o sin una cierta parte del texto.

```
pattern = re.compile(r"colou?r")
text1 = "My favorite color is blue."
text2 = "My favourite colour is blue."

print(pattern.search(text1).group())  # color
print(pattern.search(text2).group())  # colour
```

Coincidencia cero o más veces

El carácter * en una expresión regular significa “cero o más veces”. Es decir, el patrón que lo precede puede no aparecer o repetirse muchas veces. Es una de las formas más flexibles de indicar repetición.

```
pattern = re.compile(r"lo*l")

print(pattern.match("ll"))         # match
print(pattern.match("lol"))        # match
print(pattern.match("loool"))      # match
print(pattern.match("loooooool"))  # match
```

El símbolo + en una expresión regular significa "uno o más", es decir, el patrón que lo precede debe aparecer al menos una vez para que haya una coincidencia. A diferencia del *, que permite que el patrón no aparezca, el + exige que esté presente mínimo una vez.

```
pattern = re.compile(r"lo+l")

print(pattern.match("lol"))        # match
print(pattern.match("loool"))      # match
print(pattern.match("looooooool"))  # match
print(pattern.match("ll"))         # None
```

Coincidencia de repeticiones específicas con corchetes

Los corchetes {} en una expresión regular se utilizan para especificar un número exacto de repeticiones de un patrón. Esto te permite definir cuántas veces debe aparecer un grupo o carácter dentro de la cadena para que haya una coincidencia.

La sintaxis es:

- {n}: Coincide exactamente con n repeticiones del patrón.
- {n,m}: Coincide con al menos n y como máximo m repeticiones del patrón.
- {n,}: Coincide con al menos n repeticiones del patrón.

```
# Número específico de repeticiones
pattern = re.compile(r"\d{3}")
match = pattern.match("123")
print(match.group())  # 123

# Rango de repeticiones
pattern = re.compile(r"\d{2,4}")
match = pattern.match("4567")
print(match.group())  # 4567

# Al menos un número de repeticiones
pattern = re.compile(r"\d{2,}")
match = pattern.match("12345")
print(match.group())  # 12345
```

Coincidencias codiciosas y no codiciosas

En las expresiones regulares de Python, la coincidencia codiciosa y no codiciosa determinan cómo se emparejan las cadenas cuando hay opciones ambiguas. Por defecto, las expresiones regulares en Python son codiciosas, lo que significa que intentarán hacer coincidir la mayor cantidad de texto posible, incluso si hay varias opciones válidas. En contraposición, la coincidencia no codiciosa busca el texto más corto posible que cumpla con el patrón.

Coincidencia Codiciosa (Greedy)

Por ejemplo, si utilizas la expresión regular (Ha){3,5} en el texto 'HaHaHaHaHa', Python tratará de hacer coincidir tantas repeticiones como sea posible, es decir, 5 instancias de Ha:

```
pattern = re.compile(r"(Ha){3,5}")
match = pattern.search("HaHaHaHaHa")
print(match.group())  # HaHaHaHaHa
```

En este caso, la coincidencia codiciosa devuelve 'HaHaHaHaHa' porque intenta encontrar la longest coincidencia posible que cumpla con las condiciones, es decir, 5 repeticiones de 'Ha'.

Coincidencia No Codiciosa (Non greedy)

Si deseas que la expresión regular encuentre la coincidencia más corta posible en lugar de la más larga, puedes hacer que el patrón sea no codicioso utilizando el signo de interrogación (?) después de las llaves. Esto indica que la expresión debe buscar el mínimo número de repeticiones posibles, no el máximo. Por ejemplo, usando la versión no codiciosa de la expresión (Ha){3,5}?:

```
pattern = re.compile(r"(Ha){3,5}?")
match = pattern.search("HaHaHaHaHa")
print(match.group())  # HaHaHa
```

En este caso, la expresión regular devuelve 'HaHaHa' porque es la coincidencia más corta posible que cumple con el patrón (Ha){3,5} (es decir, 3 repeticiones de 'Ha').

Método findall()

El método findall() en Python es parte de los objetos Regex y se utiliza para encontrar todas las coincidencias de un patrón en una cadena. A diferencia de search(), que devuelve un único objeto de coincidencia (Match) correspondiente a la primera coincidencia, findall() devuelve una lista con todas las cadenas que coinciden con el patrón.

```
# Definir la expresión regular para un número de teléfono
pattern = re.compile(r'\d{3}-\d{3}-\d{4}')

# Texto en el que buscar
text = "Aquí están los números: 415-555-1234, 408-555-5678, 650-555-9999."

# Usar findall() para obtener todas las coincidencias
matches = pattern.findall(text)

print(matches) # ['415-555-1234', '408-555-5678', '650-555-9999']
```

Devuelve una lista con todas las coincidencias encontradas en el texto que se pasa como argumento. Cada coincidencia será una cadena que cumple con el patrón de la expresión regular. Si el patrón contiene grupos, findall() devuelve una lista de tuplas, donde cada tupla contiene los grupos coincidentes.

Clases de caracteres

Las clases de caracteres son una forma de agrupar caracteres para que coincidan con un conjunto de ellos. Esto permite escribir patrones más generales y flexibles. Un ejemplo básico de clase de caracteres es \d, que representa cualquier dígito numérico (es decir, cualquier número del 0 al 9).

\d – Dígitos numéricos

Representa cualquier carácter que sea un dígito numérico.La expresión regular \d{3} coincidirá con cualquier secuencia de tres dígitos consecutivos, como 415, 555, 123.

\w – Caracteres de palabra

Coincide con cualquier carácter que sea una letra, un número o un guión bajo (_). Esto incluye letras mayúsculas y minúsculas, números y el carácter _. La expresión regular \w+ coincidirá con cualquier secuencia de caracteres alfanuméricos, como abc123, Hello_World, o variable_1.

\s – Espacios en blanco
Coincide con cualquier tipo de espacio en blanco, que incluye espacios, tabulaciones, saltos de línea y otros caracteres de espacio. La expresión regular \s+ coincidirá con uno o más espacios en blanco, lo que podría incluir espacios, tabulaciones o saltos de línea.

\D – No dígitos
Coincide con cualquier carácter que no sea un dígito numérico. Es el opuesto de \d. La expresión regular \D coincidirá con cualquier carácter que no sea un número, como letras o símbolos.

\W – No caracteres de palabra
Coincide con cualquier carácter que no sea una letra, un número o un guión bajo. Es el opuesto de \w. La expresión regular \W coincidirá con caracteres especiales como !, @, #, o símbolos de puntuación.

\S – No espacio en blanco
Coincide con cualquier carácter que no sea un espacio en blanco, es decir, cualquier carácter que no sea un espacio, tabulación o salto de línea. La expresión regular \S coincidirá con caracteres como letras, números, o símbolos, pero no con espacios.

[ ] – Conjunto de caracteres
Dentro de corchetes [], puedes definir un conjunto de caracteres con los que deseas hacer coincidir. Se pueden incluir rangos (por ejemplo, a-z para letras minúsculas) y se pueden combinar varios tipos de caracteres. La expresión regular [a-z] coincidirá con cualquier letra minúscula, mientras que [A-Za-z] coincidirá con cualquier letra, ya sea mayúscula o minúscula.

[^] – Negación de un conjunto de caracteres
Si colocas un ^ al principio de un conjunto de caracteres dentro de los corchetes, esto significa que quieres hacer coincidir todo lo que no esté en el conjunto. La expresión regular [^0-9] coincidirá con cualquier carácter que no sea un dígito numérico.

$ - Final de una línea
El signo de dólar ($) en las expresiones regulares tiene un significado especial. Se utiliza para indicar el final de una línea o cadena. El patrón sólo coincidirá si el texto termina exactamente como se indica antes del $. Es útil para verificar si una cadena finaliza con cierta palabra, número o símbolo

. - Carácter comodín
El carácter comodín . (punto) en expresiones regulares se conoce como wildcard y tiene una función muy poderosa:

Coincidencia sin importar mayúsculas o minúsculas

Normalmente, las expresiones regulares distinguen entre mayúsculas y minúsculas.

```
robocopRegex = re.compile(r'robocop', re.IGNORECASE)
resultado = robocopRegex.findall('RoboCop protects the innocent. ROBOCOP is strong. robOcop is cool.')
print(resultado) # ['RoboCop', 'ROBOCOP', 'robOcop']
```

Sustituyendo cadenas con el método sub()

Las expresiones regulares no solo se usan para buscar patrones, sino también para reemplazarlos por otros textos. Para esto, Python ofrece el método sub() de los objetos Regex.

```
namesRegex = re.compile(r'Agent \w+')
resultado = namesRegex.sub('CENSORED', 'Agent Alice gave the documents to Agent Bob.')
print(resultado) # CENSORED gave the documents to CENSORED.
```

Sustitución parcial usando \1

También podés usar grupos para mantener partes del texto original. Por ejemplo, para solo censurar el apellido del agente:

```
agentNamesRegex = re.compile(r'Agent (\w)\w*')
resultado = agentNamesRegex.sub(r'Agent \1***', 'Agent Alice told Agent Bob that Agent Charlie was late.')
print(resultado) #Agent A*** told Agent B*** that Agent C*** was late.
```

Manejo de expresiones regulares complejas

Las expresiones regulares funcionan muy bien con patrones de texto simples, pero cuando los patrones se vuelven más complicados, las regex pueden volverse largas, difíciles de leer y de mantener Python permite que las regex se escriban en múltiples líneas, con espacios y comentarios, para mejorar la legibilidad. Esto se activa al compilar usando el flag re.VERBOSE.

```
phoneRegex = re.compile(r'''
    (\d{3}|\(\d{3}\))?      # código de área (opcional)
    (\s|-|\.)?              # separador (espacio, guión o punto)
    \d{3}                   # primeros 3 dígitos
    (\s|-|\.)               # separador
    \d{4}                   # últimos 4 dígitos
    (\s*(ext|x|ext.)\s*\d{2,5})?  # extensión (opcional)
''', re.VERBOSE)
```

Parte III: Leer y escribir archivos

Archivos y Rutas de Archivos

En informática, un archivo tiene dos propiedades fundamentales: el nombre del archivo y la ruta del archivo. El nombre es el identificador del archivo, como por ejemplo projects.docx. Este nombre suele contener una extensión, que es la parte que aparece después del último punto (.). La extensión indica el tipo de archivo, como .docx para documentos de Word, .txt para archivos de texto, o .jpg para imágenes.

Por otro lado, la ruta del archivo especifica la ubicación del archivo dentro del sistema del computador. Por ejemplo, en una computadora con Windows, una ruta típica podría verse así: C:\Users\asweigart\Documents\projects.docx. Esta ruta nos dice que el archivo projects.docx se encuentra dentro de la carpeta Documents, que está dentro de la carpeta asweigart, la cual, a su vez, está dentro de la carpeta Users.

Cada carpeta (también llamada directorio) puede contener archivos y otras carpetas. Esto permite organizar la información en una estructura jerárquica, facilitando el acceso y la gestión de los archivos en el sistema operativo.

Crear rutas

Crear rutas de archivos de manera segura y compatible entre distintos sistemas operativos es sencillo usando la función os.path.join() del módulo os. Esta función recibe como argumentos las distintas partes de una ruta (por ejemplo, nombres de carpetas y archivos) y devuelve una cadena con la ruta completa, utilizando los separadores adecuados para el sistema operativo en el que se está ejecutando el programa.

Directorio de trabajo actual

Cualquier nombre de archivo o ruta que no comience con la carpeta raíz se asume que está ubicado dentro del directorio de trabajo actual. Este directorio es la carpeta desde la que se ejecutan los comandos o scripts de Python, y puedes obtener su ruta con la función os.getcwd(). Además, puedes cambiar este directorio con la función os.chdir().

```
os.getcwd() # 'C:\\Python34'

os.chdir('C:\\Windows\\System32')
os.getcwd() # 'C:\\Windows\\System32'
```

Rutas absolutas y Rutas relativa

Cuando trabajás con archivos en Python (o en cualquier sistema operativo), hay dos formas principales de especificar una ruta:

Ruta absoluta: Siempre comienza desde la carpeta raíz del sistema. Es decir, define el camino completo hasta el archivo o carpeta, sin importar desde dónde se ejecute tu programa. Ej: C:\Users\NombreUsuario\Documentos\archivo.txt

Ruta relativa: Se define en relación con el directorio de trabajo actual de tu programa. Es más corta y útil si los archivos están en la misma estructura de carpetas del proyecto. Ej: documentos/archivo.txt

El módulo os.path

https://docs.python.org/3/library/os.path.html

El módulo os.path de Python contiene muchas funciones útiles para trabajar con nombres de archivos y rutas. Es especialmente valioso porque permite manejar rutas de una forma compatible con cualquier sistema operativo (Windows, macOS, Linux, etc.).

Por ejemplo, ya viste el uso de os.path.join(), que construye rutas correctamente sin preocuparse por los separadores (\ en Windows, / en Unix). Esta herramienta te permite escribir programas que funcionen bien en cualquier sistema y que manejen archivos de forma robusta y flexible.

Manejo de rutas absolutas y relativas

El módulo os.path de Python ofrece funciones útiles para manejar rutas absolutas y relativas. Permite convertir una ruta relativa en su equivalente absoluta usando os.path.abspath(), lo cual resulta útil para obtener la ubicación completa de un archivo o carpeta, independientemente de dónde se ejecute el programa. También se puede verificar si una ruta ya es absoluta utilizando os.path.isabs(), que devuelve True si la ruta proporcionada comienza desde el directorio raíz del sistema de archivos. Estas funciones facilitan la navegación y gestión de archivos dentro de los scripts, asegurando que las rutas sean interpretadas correctamente en distintos entornos.

También proporciona funciones para dividir una ruta en sus componentes. Con os.path.dirname(), se puede obtener la parte del directorio de una ruta, es decir, todo excepto el nombre del archivo. Por otro lado, os.path.basename() devuelve solo el nombre del archivo. Esto es útil cuando se necesita trabajar con diferentes partes de una ruta por separado, por ejemplo, para organizar archivos o generar rutas dinámicamente.

Permite verificar si una ruta realmente existe en el sistema de archivos. La función os.path.exists(path) devuelve True si el archivo o carpeta existe, y False si no. Además, se pueden usar funciones como os.path.isfile(path) para saber si la ruta corresponde a un archivo, o os.path.isdir(path) para verificar si corresponde a una carpeta. Estas comprobaciones son esenciales antes de intentar abrir archivos o modificar directorios para evitar errores en la ejecución del programa.

El método os.path.split() hará que te devuelva cada componente del path en una lista.

```
ruta = 'C:/Windows/System32'
print(os.path.exists(ruta)) # True si existe, False si no

archivo = 'C:/Windows/System32/notepad.exe'
print(os.path.isfile(archivo)) # True si es archivo, False si no

carpeta = 'C:/Windows/System32'
print(os.path.isdir(carpeta)) # True si es carpeta, False si no

ruta_completa = 'C:/Users/Usuario/Documentos/proyecto/final.py'

directorio, archivo = os.path.split(ruta_completa)

print("Directorio:", directorio)
print("Archivo:", archivo)
```

Encontrar tamaño y lista de archivos y carpetas

Una vez que manejás correctamente las rutas de archivos, podés empezar a obtener información específica sobre archivos y carpetas. El módulo os.path de Python ofrece funciones útiles para encontrar el tamaño de un archivo (en bytes) y para listar los archivos y carpetas que hay dentro de una carpeta determinada. Esto es particularmente útil cuando estás escribiendo scripts que necesitan analizar o gestionar contenido del sistema de archivos.

Por ejemplo, para obtener el tamaño de un archivo, podés usar os.path.getsize(). Y para listar el contenido de una carpeta, podés usar os.listdir().

```
archivo = 'C:/Users/Usuario/Documents/ejemplo.txt'
tamaño = os.path.getsize(archivo)

print(f"El tamaño del archivo es: {tamaño} bytes")

carpeta = 'C:/Users/Usuario/Documentos'
contenido = os.listdir(carpeta)

print("Contenido de la carpeta:")
for elemento in contenido:
    print(elemento)
```

Validez de ruta

Antes de trabajar con rutas de archivos o carpetas, es importante verificar si existen, ya que muchas funciones en Python generarán errores si se les pasa una ruta inválida. Para evitar esto, el módulo os.path proporciona funciones que permiten verificar si una ruta existe, y también si esa ruta corresponde a un archivo o a una carpeta.

```
ruta = 'C:/Users/Usuario/Documentos/ejemplo.txt'

# Verificar si la ruta existe
if os.path.exists(ruta):
    print("La ruta existe.")
else:
    print("La ruta no existe.")

# Verificar si el archivo existe
if os.path.isfile(ruta):
    print("Es un archivo.")
else:
    print("No es un archivo.")

# Verificar si la carpeta existe
if os.path.isdir(ruta):
    print("Es una carpeta.")
else:
    print("No es una carpeta.")
```

Parte IV: Organización de archivos

El módulo shutil (shell utilities)

El módulo shutil (abreviatura de shell utilities) en Python permite realizar operaciones de alto nivel con archivos y carpetas, como copiar, mover, renombrar o eliminar. Este módulo es especialmente útil cuando querés automatizar tareas de gestión de archivos en tus programas.