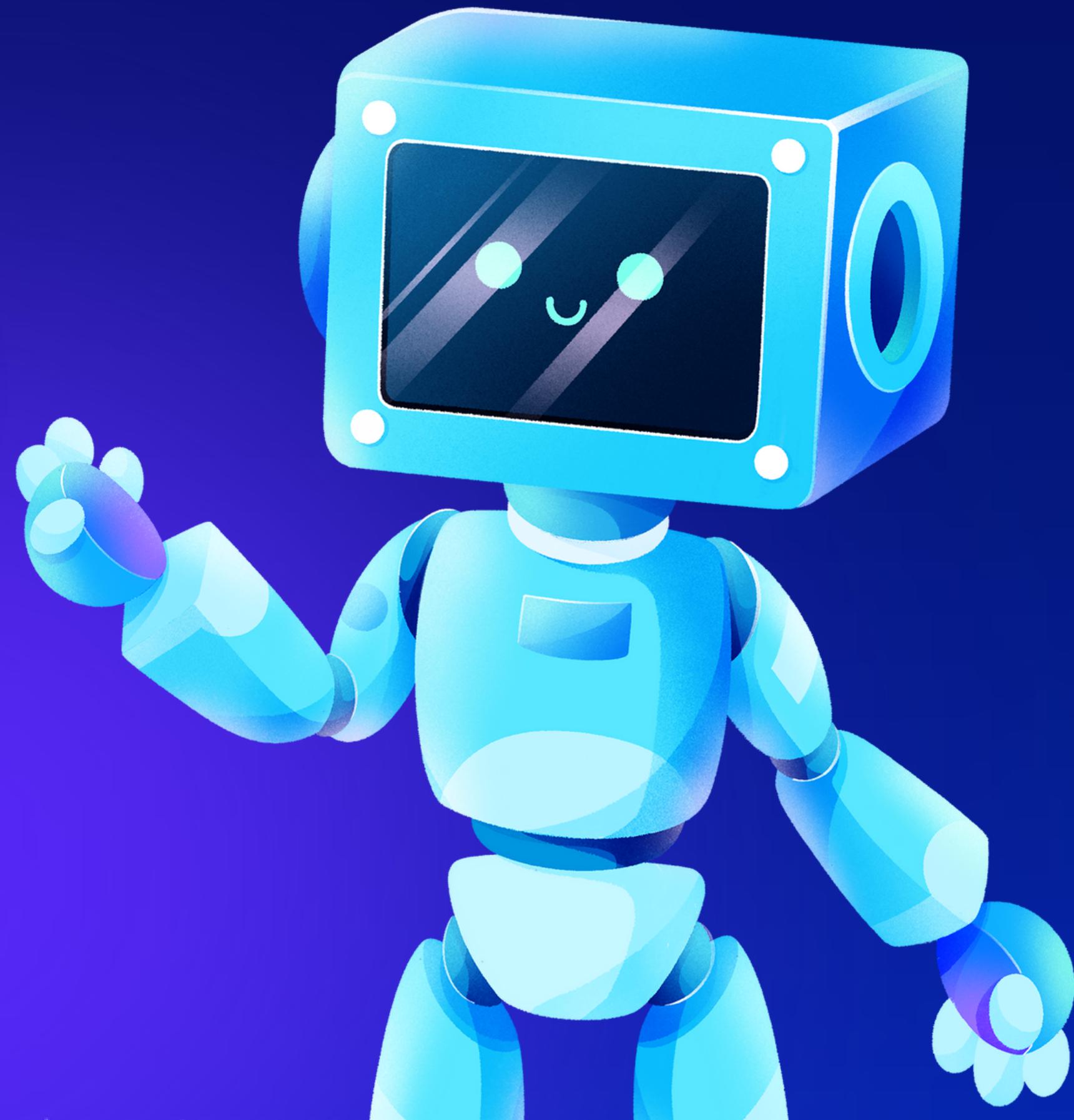


ALGORITMOS 2

UNIDAD 1

Licenciatura en Ciencias de Datos





INDICE

- | | |
|----------------------------------|----|
| • Paradigma Funcional | 01 |
| • Funciones: Primera Clase | 02 |
| • Funciones: Orden Superior | 03 |
| • Funciones: Puras | 04 |
| • Funciones: Composición | 05 |
| • Funciones: Lambda | 06 |
| • Structural Pattern Matching | 07 |
| • Evaluación estricta y perezosa | 08 |
| • Inmutabilidad en Python | 09 |
| • Actividades | 10 |





STRUCTURAL PATTERN MATCHING

Definición

Es un heramienta del estilo Switch Case, pero mucho mas poderosa.

```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something's wrong with the Internet"
```

OBJETIVO: retornar el error.

Aquí vemos:

- “match”: define sobre qué variable queremos hacer la comparación
- “case” : define las posibles alternativas de forma similar a un “switch” clásico.

Anteriormente esto podría implementarse en Python con una secuencia de “if..elif..else” pero de este modo queda más elegante.

NOTA: “_” actúa como “wildcard” o comodín para encajar con cualquier expresión que no encaje con el resto de opciones.



STRUCTURAL PATTERN MATCHING

Definición

Es un heramienta del estilo Switch Case, pero mucho mas poderosa.

```
match my_list:  
    case []:  
        print("Lista vacía")  
    case [x]:  
        print(f"Lista de un elemento: {x}")  
    case [1, 2] | [2, 1] | [1, 3] | [3, 1]:  
        print(f"Estas combinaciones me interesan mucho")  
    case [x, y]:  
        print(f"Lista con dos elementos: {x} y {y}")  
    case [x, y, z]:  
        print(f"Lista con tres elementos: {x}, {y} y {z}")  
    case [0, 1, 1, 2, *tail]:  
        print(f"Parece que es la serie de Fibonacci...")  
    case ["end", "of", "game"]:  
        print(f"Se acabó el juego...")  
    case [x, y, *tail]:  
        print(f"Lista con más de tres elementos. Los dos  
primeros son: {x} y {y}")
```

OBJETIVO: match con diferentes estructuras y tipos de datos

En este ejemplo vemos como junto al case, podemos no solo poner literales o enteros, sino listas con diferente estructura. De esta forma podemos hacer un manejo muy cómodo y sencillo en función de la estructura y contenido de la lista recibida y ejecutar diferente lógica, sin la necesidad de usar la función len() ni acceder explícitamente a los elementos de la lista.

NOTA: Como vemos en el tercer "case", podemos usar el operador "|", (OR) para declarar varias opciones que harían match en ese caso.



STRUCTURAL PATTERN MATCHING

```
match customer_data:  
    case {"password": password, **personal_data}:  
        customer.set_password(password=password)  
        customer.update_personal_data(personal_data)  
    case {"gdpr_check": True, "customer_id": cid}:  
        apply_gdpr_policies(cid)  
    case dict(x) if not x:  
        raise Exception("no data to process")
```

OBJETIVO: match con diccionarios

En este ejemplo vemos diferentes casos: en el primer case estamos verificando la presencia de una clave en concreto en el diccionario; en el segundo case, de una clave y un valor específicos; y en el tercero, si se trata de un diccionario vacío.

```
match event.get():  
    case Click(position=(x, y)):  
        handle_click_at(x, y)  
    case KeyPress(key_name="Q") | Quit():  
        game.quit()  
    case KeyPress(key_name="up arrow"):  
        game.go_north()  
    ...  
    case KeyPress():  
        pass # Ignore other keystrokes  
    case other_event:  
        raise ValueError(f"Unrecognized event: {other_event}")
```

OBJETIVO: match con objetos

Para ello, en el case declararemos la construcción del objeto con los argumentos correspondientes nombrados y de esta forma haremos matching.

NOTA: ahorrarnos un montón de llamadas a la función isinstance()



STRUCTURAL PATTERN MATCHING

```
from dataclasses import dataclass

@dataclass
class Pair:
    first: int
    second: int

pair = Pair(10, 20)

match pair:
    case Pair(0, x):
        print("Case #1")
    case Pair(x, y) if x == y:
        print("Case #2")
    case Pair(first=x, second=20):
        print("Case #3")
    case Pair as p:
        print("Case #4")
```

OBJETIVO: match con objetos

También es posible no nombrar los argumentos del constructor y pasarlos de forma posicional, pero para ello tendremos que apoyarnos en el decorador [dataclass](#) de la biblioteca estándar de Python o definiendo el atributo [match_args](#) de cualquiera de nuestras clases

NOTA: en este último ejemplo observamos también la posibilidad de añadir “guardas” o condicionales adicionales al matching. En el caso 2 vemos cómo se ha añadido una condición adicional y es que x sea igual a y a través de una construcción “if”. Esto nos permite llegar a un nivel de control muy fino.



ESTRATEGIAS DE EVALUACIÓN

Se computa una expresión a través de su reescritura hasta llegar a expresiones irreducibles (forma normal o canónica)

Existen dos órdenes:

- 1) Orden aplicativo: se reducen primero las expresiones reducibles más internas, no contienen términos reducibles
- 2) Orden normal: se reducen primero las expresiones reducibles más externas.

```
def cuadrado(x):  
    return x * x  
  
cuadrado(4 + 2)
```

Orden aplicativo: cuadrado(4+2)->cuadrado(6) -> 6*6 -> 36
Orden normal: cuadrado(4+2) -> (4+2) * (4+2) -> 6*6 ->36



EVALUACIÓN ESTRICTA Y PEREZOSA

Estricta

Es el método de evaluación por defecto en la mayoría de los lenguajes de programación. Se evalúan primero las subexpresiones más internas, similar al orden aplicativo. Se suele relacionar con la evaluación impaciente donde una expresión se evalúa tan pronto como se encuentra durante el proceso de ejecución. Debemos evaluar todas las expresiones, aún si no fueran necesarias para calcular el valor.

EN PYTHON CASI TODO SE EVALÚA DE FORMA ESTRICTA

Ventajas

- Es fácil de entender y usar.
- No hay posibilidad de olvidarse de cargar los datos necesarios.

Desventajas

- Puede llevar a un uso innecesario de recursos si se cargan datos que no se van a usar.
- Puede causar problemas de rendimiento si se cargan grandes cantidades de datos.

Perezosa

Es el método de evaluación que retrasa el cálculo de una expresión hasta que su valor sea necesario. Es fundamental en programación funcional ya que permite trabajar con "estructuras infinitas" (hasta el límite del tipo)

Ventajas

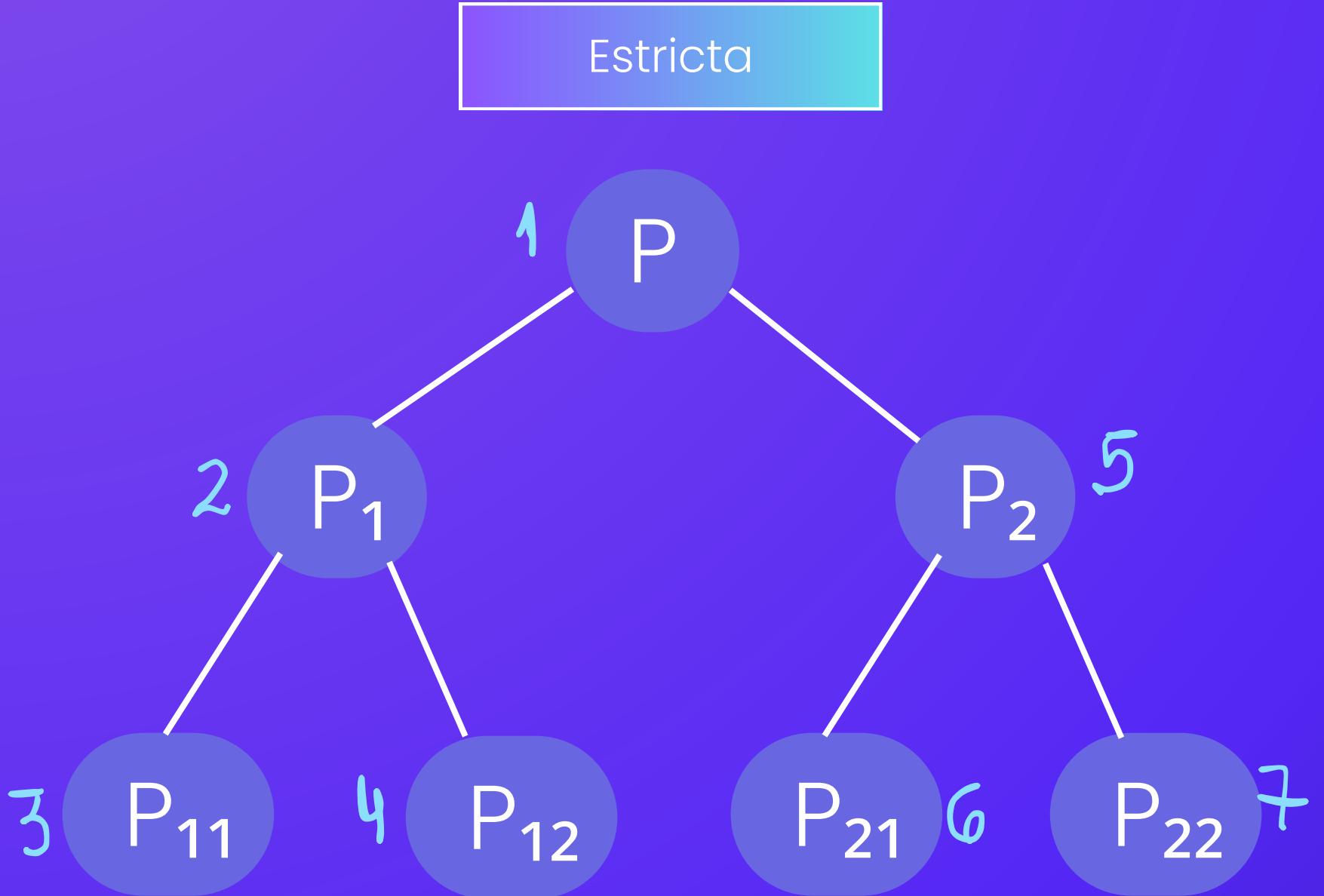
- Puede mejorar el rendimiento al evitar cálculos innecesarios.
- Puede reducir el consumo de memoria al crear valores solo cuando se necesitan.

Desventajas

- Puede ser más difícil de entender y usar.
- Puede causar problemas si se olvida de cargar los datos necesarios.

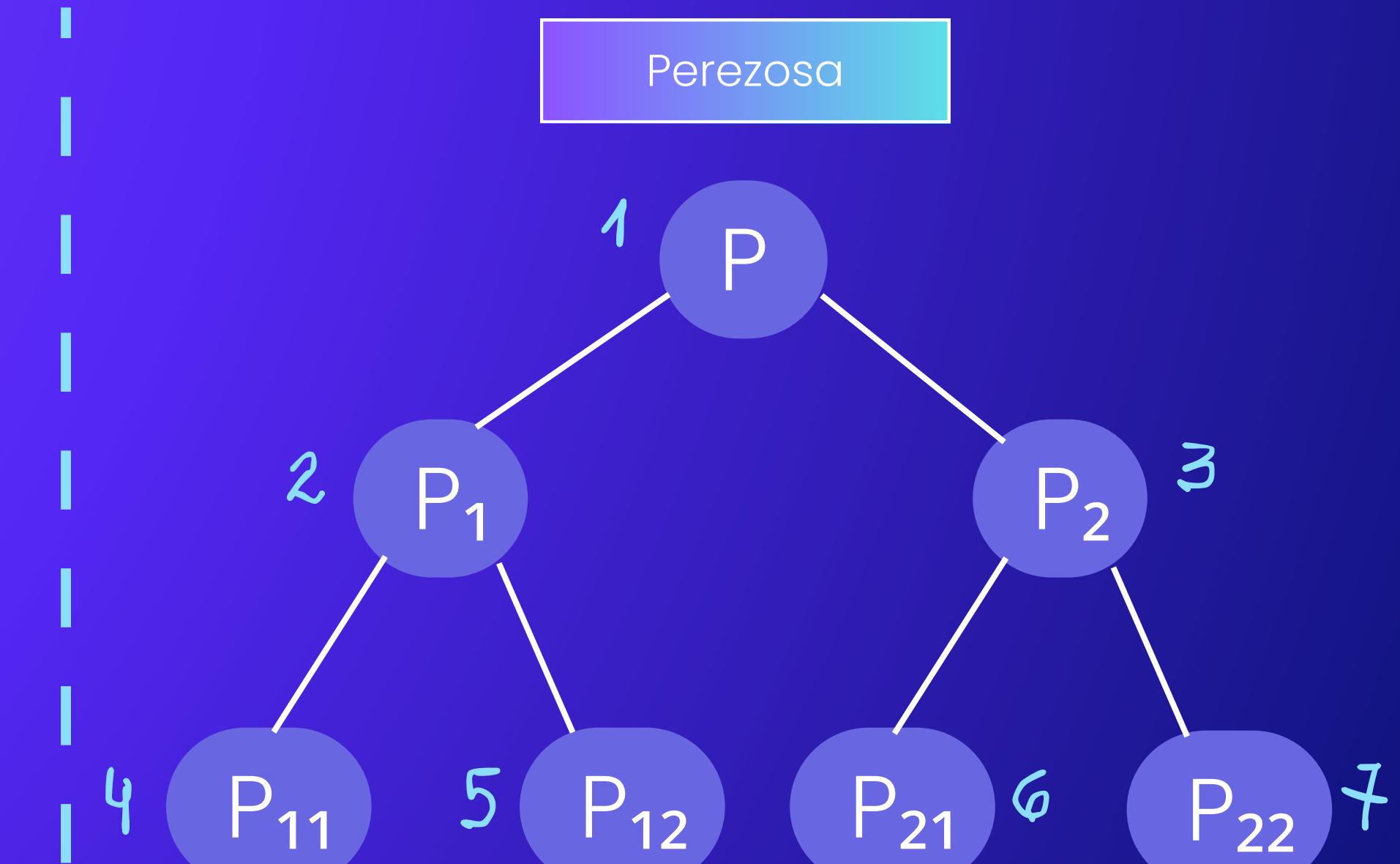


EVALUACIÓN ESTRICTA Y PEREZOSA



- Profundidad Primero
- Ramas Primero

Mejores Soluciones Primero



- Ancho Primero
- Límites Primero

Mejores Límites Primero



INMUTABILIDAD EN PYTHON

Definición

En el contexto de la programación, una variable es inmutable cuando su valor no se puede modificar. Y un objeto lo es cuando su estado no puede ser actualizado tras la creación del objeto.

En python podemos ver ejemplos de objetos mutables e inmutables

```
# Create a list
numbers = [1, 2, 3]
print(id(numbers)) # Output: 4390459520 (example id)

# Modify the list
numbers[0] = 10
print(id(numbers)) # Output: 4390459520 (same id, so the list is still the same object)
print(numbers) # Output: [10, 2, 3]
```

Los **objetos mutables** son aquellos cuyo contenido puede modificarse luego de haber sido creados.
Por ejemplo: listas, diccionarios, conjuntos (sets).



INMUTABILIDAD EN PYTHON

```
# Create an integer
num = 10
print(id(num)) # Output: 4390459520 (example id)

# Try to modify the integer
num = 20
print(id(num)) # Output: 4390459528 (different id, so the integer is a new object)
print(num) # Output: 20
```

Los **objetos inmutables** son aquellos cuyo contenido no puede modificarse luego de haber sido creados.

Por ejemplo: Números, Bytes, Cadenas, Tuplas, Booleanos.