Doctorado en Tecnologías de Transformación Digital

Materia: Ingeniería para el procesamiento masivo de datos

Hoja de Trucos de Python Avanzado

Alumno: Luis Alejandro Santana Valadez

Tutor: Dr. Jonás Velasco Álvarez

Pachuca de Soto, Hgo.
06 de julio de 2025

1. Compresores y Generadores

map(). Aplica una función a cada elemento de un iterable.

```
squared = list(map(lambda x: x**2, range(5))
    ↪ )
combined = list(map(lambda *args: sum(args),
    ↪  range(3), [10, 20, 30]))
```

zip(). Agrupa elementos de múltiples iterables.

```
nums = [1, 2, 3]
chars = ['a', 'b', 'c']
result = list(zip(nums, chars))
```

filter(). Filtra elementos según una función booleana.

```
nums = [2, 5, 8, 1]
res = list(filter(lambda x: x > 4, nums))
```

List Comprehensions. Construcción compacta de listas.

```
[x**2 for x in range(5)]
[x*y for x in range(1, 4) for y in range(1,
    ↪ x+1)]
```

Dictionary Comprehensions. Construcción de diccionarios.

```
{c: ord(c) for c in 'abc'}
```

Generator Functions. Generadores con `yield`.

```
def infinite_count(start=0):
    while True:
        yield start
        start += 1
```

Generator Expressions. Sintaxis compacta para generadores.

```
(x**2 for x in range(5))
```

2. Programación Orientada a Objetos (P.O.O.)

Clases y Objetos.

```
class Persona:
    especie = 'Humano'

juan = Persona()
juan.nombre = "Juan"
```

Constructor __init__.

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

Herencia y super().

```
class Base:
    def __init__(self):
        print("Base init")

class Derivada(Base):
    def __init__(self):
        super().__init__()
```

@staticmethod y @classmethod.

```
class Util:
    @staticmethod
    def suma(a, b): return a + b
```

@property y @cached_property.

```
class Persona:
    def __init__(self, edad):
        self._edad = edad

    @property
    def edad(self): return self._edad
```

Sobrecarga de Operadores.

```
class Contador:
    def __init__(self, val): self.val = val
    def __add__(self, otro): return Contador
        ↪ (self.val + otro.val)
```

3. Decoradores e Iteradores

Decoradores con argumentos.

```
def decorador(thresh):
    def wrapper(func):
        def inner(*args, **kwargs):
            result = func(*args, **kwargs)
            if result > thresh:
                print(" Muy  grande!")
            return result
        return inner
    return wrapper
```

Iteradores personalizados.

```
class Contador:
    def __init__(self): self.i = 0
    def __iter__(self): return self
    def __next__(self):
        self.i += 1
        if self.i > 5: raise StopIteration
        return self.i
```

4. Visualización con Matplotlib

```
import matplotlib.pyplot as plt
plt.plot([1, 2, 3], [4, 5, 6])
plt.title("Gr fica de l nea")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

```
plt.bar(['A', 'B', 'C'], [10, 20, 15])
plt.title('Gr fico de barras')
plt.show()
```

5. Visualización con Seaborn

```
import seaborn as sns
import pandas as pd

data = pd.DataFrame({
    'Edad': [22, 35, 58, 44],
    'Salario': [22000, 34000, 56000, 48000],
    'G nero': ['M', 'F', 'M', 'F']
})

sns.scatterplot(x="Edad", y="Salario", hue="G nero", data=data)
```