FUNCIONES Y OOP

TEMARIO

1. Iteradores
2. Funciones
    1. Estructura
    2. Argumentos
    3. `return`
    4. `*args`
    5. `**kwargs`

3. OOP
    1. Clases
    2. Métodos
    3. `self`
    4. Constructor `__init__`
    5. Sobrecarga de operadores

1. ITERADORES

Los iteradores son objetos que contienen un número contable de valores. Son objetos que pueden ser iterados para recorrer todos los valores de los mismos.

- `range()`

2. FUNCIONES

Las funciones son bloques de código reutilizable generalmente asociado a un nombre o *identificador*.
Ejemplos de ellas son:

- `print()`
- `input()`

ESTRUCTURA

Para declara una función escribimos `def` seguido del *nombre* a asignarle a su vez seguido paréntesis y del caracter de inicio de bloque (`:`). El código que contiene es ahora delimitado por sangría:

```
def hello_world():
    hi = "Hello World!"
    print(hi)
```

ARGUMENTOS

Los *argumentos* pasados a una función son variables que se copian a la función para ser utilizadas por la misma. Para pasar un *parámetro* a una función debemos nombrarlo entre los paréntesis de la función.

```
def greet(name):
    print(f"Hola, {name}")
```

También se pueden definir valores *default* para los argumentos de las funciones. Estos son los valores a tomarse en caso de no recibir algún valor en esa posición:

```
def add_n(i, n=1):
    temp = i + n
```

return

La palabra reservada `return` se utiliza para *devolver* valores desde una función, para poder ser utilizados fuera de la misma.

```
my_int = 9
def add_n(i, n=1):
    temp = i + n
    return temp
```

return MÚLTIPLE

Abusando un poco de las `tuple` y *unpacking* de python, podemos hacer `return` de más de un valor a la vez:

```
def create_pepe():
    name = "José"
    age = "20"
    country = "México"
    return name, age, country
# unpacking de los valores:
nombre, edad, país = create_pepe()
```

*args

Si definimos una función que toma un argumento `*args`, agregará los *parámetros* pasados desde esa posición en adelante a una lista `args`:

```
def f(*args):
    s = 0
    for x in args:
        s += x
    return s
result = f(1, 2, 3, 4, 5)
```

**kwargs

Si definimos una función que toma un argumento **kwargs, agregará los *parámetros* nombrados pasados de ese punto en adelante a un diccionario kwargs:

```
def grades_by_subject(**kwargs):
    for subject, grade in kwargs.items():
        print(f"{subject}: {grade}")
```

OOP

La *programación orientada a objetos* es el paradigma de programación más prevalente actualmente. Consiste en modelar el problema como una serie de unidades u *objetos* con *propiedades* y *métodos* interactuando entre sí. on

CLASES

Para definir un objeto o `class` en Python, escribimos `class` seguido del *nombre* de la clase y del caracter de incio de bloque:

```
class Person:
```

MÉTODOS

Las clases pueden tener *métodos* asignadas a ellas. Estos son funciones que se ejecutan con la notación de punto (`clase.método()`). Estas son funciones declaradas dentro de la misma clase:

```
class Person:
    def say_name(self):
        print(f"Mi nombre es {self.name}")
```

`self`

En el anterior ejemplo se definió que el método `say_name(self)` toma un *argumento* `self`. Este argumento siempre es el primero. Es el argumento que representa al objeto mismo dentro de la función, es la forma que utilizamos para interactuar con las propiedades internas del método.

CONSTRUCTOR `__init__()`

El método `__init__()` es una parte esencial de toda clase. Es el método que se ejecuta cuando una clase se *instancia*. Este método generalmente se utiliza para asignar las propiedades iniciales de una clase:

```
class Person:
    def __init__(self, name=None, age=None, country=None):
        self.name = name
        self.age = age
        self.country = country
```

SOBRECARGA DE OPERADORES

En python, los operadores llaman métodos de los objetos parámetro. Esto significa que la sobrecarga de operadores se puede ejecutar de manera fácil y elegante, símplemente definiendo los métodos pertinentes del objeto en cuestión:

```
class Person:
    def __init__():
        ...
    def __add__(self, other):
        return self.age + other.age
```