Funciones PYTHON 

Funciones interesantes

str.**maketrans** y **translate**

```
testword = str.maketrans('aei' + 'aei'.upper() ,"AEI" + "AEI".lower())
```

*Lo que hace **maketrans** es hacer el translado de palabras a su CODIGO ASCII y hacer una especie de **diccionario** con la primer letra del primer parametro y la primer letra del segundo parametro, y así con todos los restantes.*

```
{97: 65, 101: 69, 105: 73, 65: 97, 69: 101, 73: 105}
a    A    e    E   i   I    A   a   E   e   I    i
```

97 es 'a' y 65 es 'A'

Esto nos va a servir para hacer la traduccion de nuestra palabra con **translate()**

```
new_word = 'cACa'.translate(testword)
print(new_word)
```

*tenemos la palabra "cACa" y vamos a usar **translate** y le vamos a pasar como parametro al diccionario que hemos creado: testword*

Lo que estamos por hacer es cuando tenemos 'A', nos pasará a: 'a'. y viceversa. Tenemos finalmente este resultado:

```
caCA
```

---

for

Formas de usarlo:

```
numeros = [1,2,3,4,5]

for num in numeros:
    if num % 2 == 0:
        print(f"{num} es par")
```

```
even_numbers = [num for num in range(21) if num % 2 == 0]
print(even_numbers)
```

Funciones Lambda

Se las conoce como ***funciones anonimas*** (porque no llevan nombre) y son una alternativa a las funciones creadas con ***def***. Se las usan normalmente dentro de funciones complejas como ***filter()*** o ***map()***.

```
numbers = [1, 2, 3, 4, 5]

even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
print(even_numbers)  # [2, 4]

squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)
```

output

```
[2, 4]
[1, 4, 9, 16, 25]
```

Excepciones (*try,except,else y finally*)

Las excepciones nos pueden servir para poder "*atrapar*", anticiparnos, manejar y responder por **errores**

```

```

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print('Division successful:', x)
finally:
    print('This block always runs.')
```

*Acá estamos haciendo una división en **try** y si por alguna razón dividimos por **0**, entra a la excepción y nos imprime el error. Con **finally** se imprime si o sí. Le estamos canchereando que este codigo siempre anda y nunca se rompe.*

```
try:
    number = int(input('Enter a number: '))
    result = 10 / number
except (ValueError, ZeroDivisionError) as e:
    print(f'Error occurred: {e}')
```

*Acá podemos hacer un grupo de excepciónes y darle un nombre **e**, e incluso imprimir el error que nos da el compilador.*

Raise

Con **raise** podemos de alguna manera "*lanzar*" nuestros propios erroes.

```
def check_age(age):
    if age < 0:
        raise ValueError('Age cannot be negative')
    return age

try:
    check_age(-5)
except ValueError as e:
    print(f'Error: {e}') # Error: Age cannot be negative
```

En este caso si la edad es menor a 0, generamos un *ValueError*

```
def parse_config(filename):
    try:
        with open(filename, 'r') as file:
            data = file.read()
            return int(data)
    except FileNotFoundError:
        raise ValueError('Configuration file is missing') from None
    except ValueError as e:
```

```
        raise ValueError('Invalid configuration format') from e

config = parse_config('config.txt')
```

Podemos tambien imprimir tanta el mensaje original como tambien agregarle el nuestro:

• En primer caso *From None* solo imprime nuestro mensaje.

• En el segundo caso *From e* ademas del mensaje, imprime el error del compilador.

Assert

Tambien tenemos *assert* que es una forma mas abreviada de *raise* y *AssertionError*

```
def calculate_square_root(number):
    assert number >= 0, 'Cannot calculate square root of negative number'
    return number ** 0.5

try:
    result = calculate_square_root(-4)
except AssertionError as e:
    print(f'Assertion failed: {e}')
```

Listas

append() y **insert()**

```
languages = ['Spanish', 'English', 'Japanese']
languages.append('Chinese')

print(languages)

languages.insert(0,'Cantonese')

print(languages)
```

output

```
#Append (final)
['Spanish', 'English', 'Japanese', 'Chinese']
#Insert index 0
['Cantonese', 'Spanish', 'English', 'Japanese', 'Chinese']
```

---

enumerate

```
languages = ['Spanish', 'English', 'Japanese']
enumerate_languages = list(enumerate(languages), start = 1)
```

output

```
[(1, 'Spanish'), (2, 'English'), (3, 'Japanese')]
```

*Como vemos, **enumerate()** nos crea una lista de tuplas con numero de orden y el valor de item de la lista.*

Otro uso:

```
for index, lang in enumerate(languages, start = 1):
    print(index,lang)
```

Zip

```
a = ("John", "Charles", "Mike")
b = ("Jenny", "Christy", "Monica", "Qoqo")

x = list(zip(a, b))

print(x)
```

*Como vemos, **zip()** es similar a **enumerate()** pero podemos tenemos que pasarle dos parámetros para que los una.*

output

```
[('John', 'Jenny'), ('Charles', 'Christy'), ('Mike', 'Monica')]
```

En este caso, vemos que "Qoqo" sobra y no lo imprime porque no puede unirlo con otro valor del primer parámetro (a)

Clases

Las clases las usamos como plantilla o modelo para crear objetos. Dentro de las clases estarán los ***atributos*** y ***los metodos*** (funciones que un objeto podrá hacer)

```
class ClassName:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def sample_method(self):
        print(self.name.upper())
```

Tenemos la clase **init** que es el iniciador del objeto. Es decir, donde vamos a inicializar nuestro objeto. **Self** hace referencia al objeto que nosotros creamos fuera de esta clase. Es decir cuando lo invocamos.

```
persona1 = ClassName("Ana", 30)

#Lo que hace la clase:
ClassName.__init__(persona1, "Ana", 30)
```

Atributos

Los atributos son variables que le perteneces al objeto. Existen dos tipos:

- **Atributos de clase** Son los datos que pertenecen a la clase, compartdo por todas las instancias. ***Podes acceder a ellas sin crear un objeto antes***!
- **Atributos de instancia** Son los datos que pertenecen **solamente** al objeto, compartdo por todas las instancias.

```
class Dog:
    species = "French Bulldog" # Class attribute

    def __init__(self, name):
        self.name = name # Instance attribute

print(Dog.species) # French Bulldog
```

Metodos

Son las funciones definidas dentro de una clase que describe el comportamiento de los objetos. (Sería un atributo de clase)

```
class Car:
    def __init__(self, color, model):
        self.color = color  # Instance attribute
        self.model = model  # Instance attribute

    def describe(self):
```

```
        return f"This car is a {self.color} {self.model}"

car_1 = Car("red", "Toyota Corolla")
car_2 = Car("green", "Lamborghini Revuelto")

print(car_1.describe()) # This car is a red Toyota Corolla
print(car_2.describe()) # This car is a green Lamborghini Revuelto
```

Ejemplos

```
class Cart:
    def __init__(self):
        self.items = []

    def add(self, item):
        self.items.append(item)

    def remove(self, item):
        if item in self.items:
            self.items.remove(item)
        else:
            print(f'{item} is not in cart')

    def list_items(self):
        return self.items

    def __len__(self):
        return len(self.items)

    def __getitem__(self, index):
        return self.items[index]

    def __contains__(self, item):
        return item in self.items

    def __iter__(self):
        return iter(self.items)
```

```
cart = Cart()
cart.add('Laptop')
cart.add('Wireless mouse')
cart.add('Ergo keyboard')
cart.add('Monitor')

for item in cart:
    print(item, end=' ') # Laptop Wireless mouse Ergo keyboard Monitor

print(len(cart)) # 4
print(cart[3]) # Monitor
```

```
print('Monitor' in cart) # True
print('banana' in cart) # False

cart.remove('Ergo keyboard')

print(cart.list_items()) # ['Laptop', 'Wireless mouse', 'Monitor']

cart.remove('banana') # banana is not in cart
```

Trabajar dinamicamente con atributos de objeto

getattr()

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person = Person('John Doe', 30)

attr_name = input('Enter the attribute you want to see: ')
print(getattr(person, attr_name, 'Attribute not found'))
```

dir() obtenemos todos los atributos

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person = Person('John Doe', 30)

# Loop through all attributes of the person object with dir() function
for attr in dir(person):
    # Ignore dunder methods like __init__ or __str__ and regular methods
    if not attr.startswith('__') and not callable(getattr(person, attr)):
        value = getattr(person, attr)
        print(f'{attr}: {value}')

# Output
# age: 30
# name: John Doe
```

setattr() podemos crear o modificar atributos

```
class Configuration:
    pass

# Data loaded at runtime (like from a config or env file)
settings_data = {
    'server_url': 'https://api.example.com',
    'timeout_sec': 30,
    'max_retries': 5
}

config_obj = Configuration()

# Dynamically set attributes using dictionary keys and values
for attr_name, attr_value in settings_data.items():
    setattr(config_obj, attr_name, attr_value)

print(config_obj.server_url) # https://api.example.com
print(config_obj.timeout_sec) # 30
```

hasattr() verifica si existe atributo (True o False)

```
    def __init__(self, name, price):
        self.name = name
        self.price = price

product_a = Product('T-Shirt', 25)

required_attributes = ['name', 'price', 'inventory_id']

for attr in required_attributes:
    if not hasattr(product_a, attr):
        print(f"ERROR: Product is missing the required attribute: '{attr}'")
    else:
        # Access the attributes dynamically once their existence is confirmed
        print(f'{attr}: {getattr(product_a, attr)}')

# Output:
# name: T-Shirt
# price: 25
# ERROR: Product is missing the required attribute: 'inventory_id'
```

delattr() Elimina atributos

```
   class UserSession:
    def __init__(self, user_id, token):
        self.user_id = user_id
        self.auth_token = token # sensitive
        self.temp_counter = 0 # temporary
```

```
session = UserSession(101, 'a1b2c3d4e5')

# List of attributes to remove dynamically before "saving" the session
attributes_to_clean = ['auth_token', 'temp_counter']

# Dynamically remove specified attributes
for attr in attributes_to_clean:
    if hasattr(session, attr):
        delattr(session, attr)
        print(f'Removed attribute: {attr}')

print('\nFinal attributes remaining:')

# Loop through the remaining attributes with dir()
for attr in dir(session):
    # Ignore dunder methods like __init__ or __str__ and regular methods
    if not attr.startswith('__') and not callable(getattr(session, attr)):
        print(f' - {attr}: {getattr(session, attr)}')

# Output:
# Removed attribute: auth_token
# Removed attribute: temp_counter

# Final attributes remaining:
#  - user_id: 101
```