Proyecto en Python

Rodrigo Vilchez Tello

December 3, 2024

Contents

1 Introduccion

- Tanto Python como C++ son dos lenguajes de programacion muy utilizados actualmente. Cada uno de estos lenguajes tiene sus propias ventajas: Python, por un lado, destaca por su versatilidad y sencillez, mientras que C++, por su eficiencia y control sobre los recursos. Siendo este el motivo de la realizacion de este proyecto.

2 Objetivos

- Desarrollar programas en python sobre 6 temas fundamentales en programacion
- Comparar la implementacion en Python con una posible solucion en C++, y analizar las principales diferencias.

3 Primera parte

3.1 Funciones

3.1.1 Factorial recursivo

Implementacion del codigo en Python:

```python
def factorial_rec(n):
    #caso base
    if n==0:
        return 1
    return n*factorial_rec(n-1)

#Ejemplo de implementacion para n!
n = int(input("Ingrese el numero: "))
fact = factorial_rec(n)
print(f"El factorial de {n} es {fact}")
```

En este caso la implementacion no difiere mucho a una posible en C++. Pero podemos notar algunas diferencias en su sintaxis, o tambien el uso de "input()" en lugar de stream.

3.1.2 Fibonacci recursivo

Implementacion del codigo en Python:

```
def fibonacci_rec(n):
    if n<=2:
        return n-1
    return fibonacci_rec(n-1)+fibonacci_rec(n-2)

#Ejemplo de implementacion para el n-simo termino
n = int(input("Ingrese la posicion: "))
n_term = fibonacci_rec(n)
print(f"El termino {n} es {n_term}")
```

Nuevamente vemos que la implementacion y logica es muy similar a la de C++. Pero es importante tener en cuenta del tipado dinamico en Python, por lo que no tenemos que ser tan explicitos con los tipos de datos a la hora de escribir codigo.

3.1.3 Hanoi recursivo

Implementacion del codigo en Python:

```
def hanoi(n,origen,auxiliar,destino):
    #caso base
    if n==1:
        print(f"Mover disco 1 de {origen} a {destino}")
        return
    #movemos los n-1 discos a auxiliar
    hanoi(n-1,origen,destino,auxiliar)
    print(f"Mover disco {n} de {origen} a {destino} ")
    #movemos finalmente los n-1 discos a destino
    hanoi(n-1,auxiliar,origen,destino)

#Ejemplo de implementacion con n discos
n = int(input("Ingrese el numero de discos: "))
hanoi(n,"Origen","Auxiliar","Destino")
```

Por ahora no vemos diferencias notables en la implementaciones comparado a C++, pero mas adelante veremos que hay algunas funcionalidades propias de Python que estan optimizadas para ciertas tareas.

3.2 Cadenas

3.2.1 Invertir una cadena

Implementacion del codigo en Python:

```python
def invertir_cad1(cadena):
    i = len(cadena) - 1
    cadena_inv = ""
    while i>=0:
        cadena_inv += cadena[i]
        i -= 1
    return cadena_inv

#usando indices negativos
def invertir_cad2(cadena):
    n = len(cadena)
    cadena_inv = ""
    for i in range(n-1,-1,-1):
        cadena_inv += cadena[i]
    return cadena_inv

def invertir_cad3(cadena):
    n = len(cadena)
    cadena_inv = ""
    for i in range(1,n+1):
        cadena_inv += cadena[-i]
    return cadena_inv

#usando slicing
def invertir_cad4(cadena):
    return cadena[ : :-1]

#Ejemplo de implementacion
cadena = input("Ingrese una cadena: ")
print(f"1. {invertir_cad1(cadena)}")
print(f"2. {invertir_cad2(cadena)}")
print(f"3. {invertir_cad3(cadena)}")
print(f"4. {invertir_cad4(cadena)}")
```

La primera funcion es una implementacion clasica, podria implementarse en C++ usando strings o arrays, en el caso de usar un array retornariamos un puntero a este.
En la segunda funcion se usa “-1” como tercer parametro en “range()” para recorrer la lista desde el ultimo elemento hasta el primero.
La tercera funcion hace uso de indices negativos para acceder a los elementos del iterable desde el ultimo elemento.

Para la cuarta funcion se hizo uso de una funcionalidad bastante util que esta optimizada en Python, llamada “slicing”, en este caso “cadena[: :-1]” nos retorna la cadena invertida.

3.2.2 Numero de palabras en una cadena

Implementacion del codigo en Python:

```
def calcular_pal1(cadena):
    n=len(cadena)
    cont = 0
    for i in range(n):
        if cadena[i].isspace():
            continue
        if cadena[i-1].isspace() or i==0:
            cont +=1
    return cont

def calcular_pal2(cadena):
    en_palabra = False
    cont = 0
    for c in cadena:
        if c.isspace():
            en_palabra = False
            continue
        if not en_palabra:
            en_palabra = True
            cont += 1
    return cont

#split() es similar a strtok() de C++
def calcular_pal3(cadena):
    lista_aux = cadena.split()
    return len(lista_aux)

#Ejemplo de implementacion
cadena = input("Ingrese una cadena: ")
print(f"1. Hay {calcular_pal1(cadena)} palabras")
print(f"2. Hay {calcular_pal2(cadena)} palabras")
print(f"3. Hay {calcular_pal3(cadena)} palabras")
```

En este caso vemos que la primera y la segunda funcion usan conceptos que pueden ser implementados en ambos lenguajes.
En la tercera funcion vemos que se usa el metodo de cadenas “split()” cuya funcion es retornar la una lista de todas las cadenas que estaban separadas de alguna forma en la cadena original. En otras palabras, una lista con todas

las palabras. De esta forma, usando la funcion “len()” a la lista obtenemos el numero de palabras. Este ultimo concepto es bastante similar a la funcion “strtok()” de C++, con la diferencia de que esta ultima trabaja en la misma cadena añadiendo caracteres nulos al final de cada palabra, y no generando una nueva estructura.

3.2.3 Verificar si una cadena es palindroma

Implementacion del codigo en Python:

```
import re

#usando expresiones regulares
def es_palindromo(cadena):
    cadena_aux = re.sub(r'[^a-zA-Z]',"",cadena)
    n = len(cadena_aux)
    for i in range(n//2):
        if cadena_aux[i]!=cadena_aux[-(i+1)]:
            return False
    return True

'''
def es_palindromo(cadena):
    cadena_aux = ("".join(cadena.split())).lower()
    n = len(cadena_aux)
    for i in range(n//2):
        if cadena_aux[i]!=cadena_aux[-(i+1)]:
            return False
    return True
'''

def es_palindromo_iter(cadena):
    cadena_aux = re.sub(r'[^a-zA-Z]',"",cadena)
    izq,der = 0,len(cadena_aux)-1
    while izq < der:
        if cadena_aux[izq]!=cadena_aux[der]:
            return False
        izq -= 1
        der -= 1
    return True

'''
def es_palindromo_iter(cadena):
    cadena_aux = ("".join(cadena.split())).lower
    izq,der = 0,len(cadena_aux)-1
    while izq < der:
        if cadena_aux[izq]!=cadena_aux[der]:
```

```
                    return False
                izq -= 1
                der -= 1
            return True
        '''

        def es_palindromo_rec(cadena):
            cadena_aux = re.sub(r'[^a-zA-Z]',"",cadena)
            if len(cadena_aux)==1:
                return True
            if cadena_aux[0]!=cadena_aux[-1]:
                return False
            return es_palindromo_rec(cadena_aux[1:-1])

        '''
        def es_palindromo_rec(cadena):
            cadena_aux = ("".join(cadena.split())).lower
            if len(cadena_aux)==1:
                return True
            if cadena_aux[0]!=cadena_aux[-1]:
                return False
            return es_palindromo_rec(cadena_aux[1:-1])
        '''

        #Ejemplo de implementacion
        cadena = input("Ingrese una cadena: ")
        print(f"1. {es_palindromo(cadena)}")
        print(f"2. {es_palindromo_iter(cadena)}")
        print(f"3. {es_palindromo_rec(cadena)}")
```

Notamos que hay una version alternativa comentada para cada funcion, estas funciones solo son validas para ignorar espacios y mayusculas al hacer la evaluacion, puesto que en la lista obtenida con "split()" y "lower()" no descartara los caracteres especiales. Es por eso que en las funciones no comentadas se usan expresiones regulares importando el modulo "re". En C++ tambien podriamos implementarlo usando la libreria "regex" y "regex_replace()" en lugar de "re.sub()".
Si no quisieramos usar expresiones regulares una alternativa seria usar el equivalente a cada caracter en su codigo ASCII. De esta forma podemos ignorar un caracter si no se encuentra entre 61 y 172 si usamos minusculas, o entre 41 y 132 si usamos mayusculas.

3.3 Referencias y asignacion dinamica

3.3.1 Intercambio de valores

Implementacion del codigo en Python:

```
#Se debe usar objetos mutables como las listas
def intercambio_var(a,b):
    a[0],b[0] = b[0],a[0]
    return

#Ejemplo de implementacion
a0 = float(input("Ingrese el valor 'a': "))
b0 = float(input("Ingrese el valor 'b': "))

a = [a0]
b = [b0]

intercambio_var(a,b)
print(f"a: {a[0]}\nb: {b[0]}")
```

Python funciona de manera distinta a C++, al trabajar con variables. Debemos saber que los datos se clasifican en mutables e inmutables. Cuando nosotros asignamos un valor inmutable a una variable, en realidad la variable no lo almacena, sino una referencia al valor, pero si asignamos un valor mutable a una variable, esta ultima sera una referencia al valor que referenciaba la otra variable. De este modo, si queremos intercambiar dos valores usando una funcion no podemos usar variables que referencian a un valor inmutable porque las variables de la funcion serian referencias al valor, por lo que solo cambiariamos la referencia de esta variable y no la original. Por eso se deben usar variables inmutables como las listas, ya que la variable de la funcion referenciara a la lista, y puede modificar sus valores. Por eso se usaron listas "a" y "b" de un solo elemento.
En C++ esto podria ser un poco mas intuitivo usando paso por referencia.

3.3.2 Suma de valores en una lista

Implementacion del codigo en Python:

```
def suma_lista(lista):
    suma = 0
    for num in lista:
        suma += num
    return suma
```

```
#Ejemplo de implementacion
numero_elem = int(input("Ingrese el numero de elementos:
    "))
lista = []
for i in range(numero_elem):
    elem = float(input(f"Ingrese el elemento {i+1}: "))
    lista.append(elem)
print(f"La suma de elementos es {suma_lista(lista)}")
```

En este caso usamos listas para almacenar una serie de numero y luego la iteramos para calcular su suma. En C++ una implementacion analoga seria usando arrays o vectores de la STL.

3.3.3 Crear matriz

Implementacion del codigo en Python:

```
def crear_matriz(m,n):
    matriz = []
    for i in range(m):
        fila = []
        for j in range(n):
            elem = float(input(f"Ingrese el elemento {i
                +1},{j+1}: "))
            fila.append(elem)
        matriz.append(fila)
    return matriz

#Ejemplo de implementacion
m = int(input("Filas: "))
n = int(input("Columnas: "))
matriz = crear_matriz(m,n)
for fila in matriz:
    print(fila)
```

Python gestiona dinamicamente la memoria, por lo que no hay necesidad de implementar operadores como “new” y “delete”. En este caso usar listas seria analogo a usar vectores de la STL en C++.

3.4 Estructuras

3.4.1 Estructura "Punto"

Implementacion del codigo en Python:

```python
from collections import namedtuple
Punto = namedtuple("Punto",["x","y"])
def distancia_punt(p1,p2):
    return ((p2.x-p1.x)**2+(p2.y-p1.y)**2)**0.5

#Ejemplo de implementacion
lista = input("Coordenadas (x,y) de p1: ").split()
x,y = int(lista[0]),int(lista[1])
p1 = Punto(x,y)
lista = input("Coordenadas (x,y) de p2: ").split()
x,y = int(lista[0]),int(lista[1])
p2 = Punto(x,y)
print(f"Distancia: {distancia_punt(p1,p2)}")
```

En Python no existen estructuras como se haria con "struct" en C++, pero podemos usar "namedtuple()" que genera un tipo especial de tupla, el primer parametro es el nombre del tipo que se manejara internamente, mientras que el nombre de la variable a la cual se le asigna es lo que usaremos nosotros para instanciar objetos. En realidad la variable es una referencia a la clase generada dinamicamente (en tiempo de ejecucion).
Esto es util para representar unicamente datos, puesto que no soporta metodos. Por eso implementamos la funcion "distancia_punt" externamente.

3.4.2 Estructura "Estudiante"

Implementacion del codigo en Python:

```python
from dataclasses import dataclass,field

@dataclass
class Estudiante:
    __nombre: str = field(default="No hay informacion")
    __edad: int = field(default=-1)
    __promedio: float = field(default=-1)
    def mostrar(self):
        print(f"Nombre: {self.__nombre}")
        print(f"Edad: {self.__edad}")
        print(f"Promedio: {self.__promedio}")
```

```
#Ejemplo de implementacion
lista = input("Ingrese los datos (nombre,edad,promedio):
    ").split()
nombre,edad,promedio = lista[0],lista[1],lista[2]
estudiante = Estudiante(nombre,edad,promedio)
estudiante.mostrar()
```

Los "dataclass" son versiones simplificadas de las clases, ya que metodos especiales como "__init__", por ejemplo, no necesitan ser implementados manualmente.

3.4.3 Estructura "Producto"

Implementacion del codigo en Python:

```
from dataclasses import dataclass,field

@dataclass
class Producto:
    __nombre: str = field(default="Sin registro")
    __precio: float = field(default=-1)
    __cantidad: int = field(default=-1)
    def mostrar(self):
        print(f"Nombre: {self.__nombre}")
        print(f"Precio: {self.__precio}")
        print(f"Cantidad: {self.__cantidad}")

#Ejemplo de implementacion
lista = input("Ingrese la informacion (nombre,precio,
    cantidad): ").split()
nombre,precio,cantidad = lista[0],lista[1],lista[2]
producto = Producto(nombre,precio,cantidad)
producto.mostrar()
```

Al igual que el anterior se definio un "dataclass" usando el decorador "@dataclass".

3.5 Archivos

3.5.1 Escribir y leer texto en un archivo

Implementacion del codigo en Python:

```
def escribir_arch(nombre_arch):
    file = open(nombre_arch,"w")
    texto = input("Ingrese el texto: ")
    file.write(texto)
    file.close()

'''
def escribir_arch(nombre_arch):
    with open(nombre_arch,"w") as file:
        texto = input("Ingrese el texto: ")
        file.write(texto)
'''

def leer_arch(nombre_arch):
    file = open(nombre_arch,"r")
    contenido = file.read()
    file.close()
    print(contenido)

'''
def leer_arch(nombre_arch):
    with open(nombre_arch,"r") as file:
       contenido = file.read()
    print(contenido)
'''

if __name__=="__main__":
    #Ejemplo de implementacion
    escribir_arch("archivo.txt")
    leer_arch("archivo.txt")
```

En Python no existen los “stream” como en C++, en cambio se define una clase file, y justamente “open()” retorna un objeto de esta clase para poder trabajar con el sistema de archivos. En este caso tanto para trabajar con texto o en formato binario, se usan las funciones “write()” y “open()”, mientras que C++ nos daba la opcion de usar los operadores de extraccion e insercion si trabajabamos con texto.
Si queremos escribir en un archivo debemos usar “w”, si quieremos leer de un archivo usaremos “r”.

3.5.2 Agregar contenido en un archivo

Implementacion del codigo en Python:

```
from archivo1 import leer_arch

def agregar_text_arch(nombre_arch):
    file = open(nombre_arch,"a")
    texto = input("Ingrese el texto: ")
    file.write(texto)
    file.close()

'''
def escribir_arch(nombre_arch):
    with open(nombre_arch,"a") as file:
        texto = input("Ingrese el texto: ")
        file.write(texto)
'''

#Ejemplo de implementacion
agregar_text_arch("archivo.txt")
leer_arch("archivo.txt")
```

Si queremos añadir contenido a un archivo usaremos “a”, caso contrario se sobreescribira el archivo con el nuevo texto ingresado.

3.6 Clases

3.6.1 Clase “Persona”

Implementacion del codigo en Python:

```
class Persona:
    def __init__(self,nombre,edad):
        self.__nombre = nombre
        self.__edad = edad
    def mostrar(self):
        print(f"Nombre: {self.__nombre}")
        print(f"Edad: {self.__edad}")

#Ejemplo de implementacion
lista = input("Ingrese los datos (nombre,edad): ").split
    ()
persona = Persona(lista[0],lista[1])
persona.mostrar()
```

Cuando definimos una clase de manera tradicional debemos implementar el contructor "__init__" manualmente.
El primer parametro debe ser una referencia al objeto, a diferencia de C++ donde la referencia o puntero "this" esta implicito. Es importante mencionar que el nombre de la referencia no necesariamente debe ser "sel", pero es buena practica nombrarlo de esa forma para hacer mas legible el codigo.

3.6.2 Clase "Cuenta bancaria"

Implementacion del codigo en Python:

```
class Cuenta_Banc:
    def __init__(self):
        self.__titular = input("Ingresa el nombre del
            titular: ")
        self.__saldo = 0
        self._id = id(self)
    def depositar(self,monto):
        self.__saldo += monto
    def retirar(self,monto):
        if monto>self.__saldo:
            print(f"No dispone de {monto} actualmente. (
                Saldo: {self.__saldo})")
        else:
            self.__saldo -= monto
    def mostrar(self):
        print(f"El titular es {self.__titular}")
        print(f"Saldo: {self.__saldo}")

#Ejemplo de implementacion
cuenta_1 = Cuenta_Banc()
cuenta_1.depositar(100)
cuenta_1.mostrar()
cuenta_1.retirar(120)
cuenta_1.mostrar()
```

Cabe resaltar que el uso de la referencia al objeto es obligatoria siempre que se necesite acceder a los atributos del objeto, caso contrario sera imposible acceder a estos.
Otro detalle importante es que si quieremos definir un atributo o metodo como privado debemos anteponer el nombre con doble guion bajo, de esta forma "__atributo".

4 Segunda parte

4.1 Generador de contraseñas

Implementacion en Python:

```
import random

def generador_cont(tam):
    contr = ""
    for i in range(tam):
        c = chr(random.choice(range(33,127)))
        contrasena += c
    return contr

#Ejemplo de implementacion
cant = int(input("Cantidad: "))
long = int(input("Longitud: "))
for _ in range(cant):
    print(generador_cont(long))
```

En este pequeño programa se importa el modulo “random” para hacer uso de la funcion “random.choice()”, que recibe como argumento una secuencia y retorna un elemento aleatorio de esta. En este caso hacemos uso del ASCII code, por eso elejimos un numero al azar entre 33 y 127, que representan caracteres imprimibles, que son convertidos a caracteres usando “chr()” para ser añadidos a la cadena que sera retornada.
En C++ la implementacion seria bastante similar, con la diferencia de que no seria necesario usar algun operador de conversion, sino que lo manejaria implicitamente.

5 Conclusiones

- Despues de realizar las distintas implementaciones sobre los temas propuestos hemos sido capaces de notar algunas de las diferencias mas importantes entre ambos lenguages. Por ejemplo, ahora sabemos que Python gestiona la memoria por si mismo y de manera dinamica, a diferencia de C++ donde esa responsabilidad cae sobre nosotros, tambien aprendimos que las variables son en realidad referencias y la asignacion depende de la mutabilidad o inmutabilidad de los valores que referencian.

6 Referencias

- Python Software Foundation. *Python Programming Language*. Enlace: `https://www.python.org/`.
- FreeCodeCamp. *25 Proyectos en Python para Principiantes*. Enlace: `https://www.freecodecamp.org/espanol/news/25-proyectos-en-python-para-princ`