Ejercicios de Python (HPBBC)

Sandra Mingo Ramírez

2024/25

Ejercicio 1: **Tabla de multiplicar.** Escribe un programa que muestre la tabla de multiplicar de un número que introduzca el usuario.

```
#Pide al usuario que introduzca un número por teclado
comprobacion = False
while comprobacion == False:
   try:
      num_usuario = int(input("Introduzca un número entero: "))
   except ValueError:
      print("El valor introducido no es un número entero. Pruebe otra vez.").
   else:
      comprobacion = True
#Utiliza un bucle for para mostrar la tabla de multiplicar
for i in range(1, 11):
   print(f"{num_usuario} x {i} = {num_usuario * i}")

#Bonus: haz el mismo ejercicio usando un bucle while
num_multiplicacion = 1

while num_multiplicacion <= 10:
   print(f"{num_usuario} x {i} = {num_usuario * i}")
   num_multiplicacion += 1
```

Ejercicio 2: **Suma de los primeros N números.** Escribe un programa que pida al usuario un número entero positivo nullN y luego calcule la suma de todos los números desde 1 hasta nullN.

```
#Pide al usuario que introduzca un número entero positivo
comprobacion = False
while comprobacion == False:
   try:
      num_usuario = int(input("Introduzca un número entero positivo: "))
   except ValueError:
      print("El valor introducido no es un número entero. Pruebe otra vez.").
      num_usuario = int(input("Introduzca un número entero: "))
   else:
      if num_usuario < 0:
         print("El número no es positivo.")
      else:
         comprobacion = True
resultado = 0
for i in range(1, num_usuario +1):
   resultado += 1
#Muestra el resultado de la suma
print(f"El resultado de la suma es {resultado}.")
```

Ejercicio 3: **Contar nucleótidos en una secuencia de ADN.** Escribe un programa que pida al usuario una secuencia de ADN y cuente cuántas veces aparece cada uno de los nucleótidos (A, T, C, G).

```
#Pide al usuario que ingrese una secuencia de ADN
secuencia_usuario = input("Ingrese una secuencia de ADN: ")
secuencia = secuencia_usuario.upper()
#Utiliza un bucle para recorrer la secuencia y contar la cantidad de veces que aparece cada
    nucleótido.
```

```
num_A = 0
num_T = 0
num_C = 0
num_G = 0
for base in secuencia:
   if base == "A":
      num_A += 1
   elif base == "T":
      num_T += 1
   elif base == "C":
      num_C += 1
   elif base == "G":
      num_G += 1
   else:
      print("Nucleótido no reconocido")
      break
#Muestra el conteo de cada nucleótido
print(f"Conteo de nucleótidos: \n A: {num_A} \n T: {num_T} \n C: {num_C} \n G: {num_G}")
```

Ejercicio 4: **Transcripción de ADN a ARN** Escribe un programa que realice la transcripción de una secuencia de ADN a ARN. En una secuencia de ARN, la base nitrogenada Timina (T) se reemplaza por Uracilo (U).

```
#Pide al usuario que ingrese una secuencia de ADN
secuencia_adn = input("Ingrese una secuencia de ADN: ")
#Utiliza un bucle para recorrer la secuencia y reemplazar todas las apariciones de T por U
bases_adn = "ATCG"
secuencia_arn = ""
for base in secuencia_adn:
   if base not in bases_adn:
      print("Secuencia no válida.")
      break
   if base == "T":
      secuencia_arn += "U"
   else:
      secuencia_arn += base
#Muestra la secuencia transcrita de ARN
print(f"Secuencia de ARN transcrita: {secuencia_arn}")
```

Ejercicio 5: **Devolución de cambio** Realiza un programa que proporcione el desglose en billetes y monedas de una cantidad entera de euros. Recuerda que hay billetes de 500, 200, 100, 50, 20, 10 y 5 € y monedas de 2 y 1 €.

```
euros_posibles = [500, 200, 100, 50, 20, 10, 5, 2, 1]
cambio = []
cantidad_devolver = int(input("Proporcione la cantidad a devolver: "))
idx = 0
while cantidad_devolver > 0:
   if cantidad_devolver - euros_posibles[idx] >= 0:
      cambio.append(euros_posibles[idx])
      cantidad_devolver -= euros_posibles[idx]
   else:
      idx += 1
print(cambio)
```

Ejercicio 6: **Intersecciones de listas** Diseña una función que reciba dos listas y devuelva los elementos comunes a ambas, sin repetir ninguno (intersección de conjuntos). Ejemplo: si recibe las listas

[1, 2, 1] y [2, 3, 2, 4], debe devolver la lista [2]. Escribe, también, otra función que reciba dos listas y devuelva los elementos que pertenecen a una o a otra, pero sin repetir ninguno (unión de conjuntos). Ejemplo: si recibe las listas [1, 2, 1] y [2, 3, 2, 4], devolverá [1, 2, 3, 4].

```
def elementos_comunes_listas(lista1, lista2):
   elementos_comunes = []
      for elemento in lista1:
         if elemento in lista2:
            elementos_comunes.append(elemento)
   return elementos_comunes

def union_elementos_listas(lista1, lista2): #No es eficiente al realizar dos bucles for.
    elementos_unidos = []
    for elemento in lista1:
      if elemento not in elementos_unidos:
         elementos_unidos.append(elemento)
    for elemento in lista2:
      if elemento not in elementos_unidos:
         elementos_unidos.append(elemento)
    return elementos_unidos

def union_elementos_listas_eficiente(lista1, lista2):
    return list(set(lista1) | set(lista2))
```

Ejercicio 7: **Modelando secuencias de ADN** Modela una clase llamada `ADN` que almacene y manipule secuencias de ADN. La clase ADN debe tener un atributo para almacenar la secuencia de nucleótidos (una cadena de caracteres compuesta por las letras A, T, G y C). Debe tener un método para contar la cantidad de cada tipo de nucleótido en la secuencia y otro que permita obtener la secuencia complementaria de ADN (A se empareja con T, C con G). Finalmente, debe tener un método que calcule la proporción de guanina y citosina (CG content).

```
class DNA:
    def __init__(self, seq):
        self.sequence = seq

    def nucleotide_count(self):
        count_A = 0
        count_T = 0
        count_C = 0
        count_G = 0
        for nucleotide in self.sequence:
            if nucleotide == 'A':
                count_A += 1
            elif nucleotide == 'T':
                count_T += 1
            elif nucleotide == 'G':
                count_G += 1
            elif nucleotide == 'C':
                count_C += 1
            else:
                print("Nucleotide not valid.")
                break
        return count_A, count_T, count_G, count_C

    def complementary_strand(self):
        complementary_strand = ''
        for nucleotide in self.sequence:
            if nucleotide == 'A':
                complementary_strand += 'T'
            elif nucleotide == 'T':
```

```
            complementary_strand += 'A'
        elif nucleotide == 'G':
            complementary_strand += 'C'
        elif nucleotide == 'C':
            complementary_strand += 'G'
        else:
            print("Nucleotide not valid.")
            break
    return complementary_strand

def cg_content(self):
    sequence_length = len(self.sequence)
    _, _, g_count, c_count = self.nucleotide_count
    return (g_count + c_count) / sequence_length * 100
```

Ejercicio 8: **Simulación de mutaciones en ADN** Expande la clase ADN del ejercicio anterior para que tenga un método que simule una mutación en una secuencia de ADN. Este método, dado un nucleótido y una posición, debe cambiar el nucleótido en esa posición. Primero se debe validar que la posición es válida. También se debe agregar un método que simule una mutación aleatoria en una posición aleatoria.

```
import random
class DNA:
    def __init__(self, seq):
        self.sequence = seq.upper()

    def get_sequence(self):
        return self.sequence
    def nucleotide_count(self):
        count_A = 0
        count_T = 0
        count_C = 0
        count_G = 0
        for nucleotide in self.sequence:
            if nucleotide == 'A':
                count_A += 1
            elif nucleotide == 'T':
                count_T += 1
            elif nucleotide == 'G':
                count_G += 1
            elif nucleotide == 'C':
                count_C += 1
            else:
                print("Nucleotide not valid.")
                break
        return count_A, count_T, count_G, count_C

    def complementary_strand(self):
        complementary_strand = ''
        for nucleotide in self.sequence:
            if nucleotide == 'A':
                complementary_strand += 'T'
            elif nucleotide == 'T':
                complementary_strand += 'A'
            elif nucleotide == 'G':
                complementary_strand += 'C'
            elif nucleotide == 'C':
                complementary_strand += 'G'
            else:
                print("Nucleotide not valid.")
                break
```

```
        return complementary_strand

    def cg_content(self):
        sequence_length = len(self.sequence)
        _, _, g_count, c_count = self.nucleotide_count
        return (g_count + c_count) / sequence_length * 100

    def mutate(self, nucleotide, position):
        if position <= len(self.sequence):
            sequence_list = list(self.sequence)
            sequence_list[position-1] = nucleotide.upper()
            self.sequence = ''.join(sequence_list)
        else:
            print("Position not valid - out of range")

    def random_mutate(self):
        nucleotides = ['A', 'T', 'G', 'C']
        random_nucleotide = random.choice(nucleotides)
        random_position = random.choice(list(range(0, len(self.sequence))))
        self.mutate(random_nucleotide, random_position)
```

Ejercicio 9: **Modelando proteínas** Modela una clase `Proteina` que almacene y manipule una secuencia de aminoácidos. Debe tener un atributo para almacenar la secuencia de aminoácidos (una cadena de letras que representan los aminoácidos: A, R, N, D, C, etc.). Implementa un método que devuelva la masa molecular aproximada de la proteína, utilizando un diccionario de masas de aminoácidos. Añade un método que permita determinar si la proteína contiene un motivo (una subcadena de aminoácidos de interés).

```
class Protein:

    molecular_weights = {'A':89.1, 'R':174.2, 'N':132.1, 'D':133.1, 'C':121.2, 'E':147.1,
        'Q':146.2, 'G':75.1, 'H':155.2, 'I':131.2, 'L':131.2, 'K':146.2, 'M':149.2, 'F':165.2,
        'P':115.1, 'S':105.1, 'T':119.1, 'W':204.2, 'Y':181.2, 'V':117.1}

    def __init__(self, aa_seq):
        self.aminoacids = aa_seq.upper()

    def protein_weight(self):
        weight = 0
        for aminoacid in self.aminoacids:
            weight += Protein.molecular_weights[aminoacid]
        return weight

    def find_motif(self, motif):
        return motif in self.aminoacids
```

Ejercicio 10: **Simulación de una célula con ADN y proteínas** Crea un sistema de clases que simule el funcionamiento básico de una célula. La célula debe contener ADN, proteínas y tener la capacidad de producir proteínas a partir del ADN. Se debe crear una clase `Cell` que contenga los objetos de las clases `DNA` y `Protein`. Esta nueva clase debe tener un método que permita transcribir una secuencia de ADN en ARN utilizando la clase ADN. La célula debe tener la capacidad de traducir ARN en proteínas, utilizando el código genético de los ejercicios anteriores, y almacenarlas. Implementa un método para mostrar todas las proteínas que la célula ha sintetizado. Agrega un método que permita realizar una mutación en el ADN y ver cómo afecta a las proteínas resultantes.

```
class Cell:
```