Procesado y Manejo de Datos Masivos

Resumen

En esta asignatura se aprenderán técnicas para procesar y manipular grandes volúmenes de datos utilizando programación y línea de comandos. Se abordará el manejo y análisis de formatos de datos comunes en bioinformática, como FASTA, GFF y GenBank, utilizando patrones que optimizan el uso de memoria y tiempo de ejecución, así como técnicas de programación paralela. Además, se realizarán operaciones avanzadas en bases de datos relacionales y no relacionales, y se explorará el acceso programático a bases de datos biomédicas online a través de sus APIs.

Sandra Mingo Ramírez

UAM - 2024/25

22 de octubre de 2024 19:11

Universidad Autónoma de Madrid
Bioinformática y Biología Computacional

Código en Github

Índice general

Capítulo I

Estrategias de programación para el parseo y extracción de datos

I.1. Formatos de ficheros bioinformáticos

Los formatos de fichero son en formato texto, y los principales son:

- FASTA: guarda secuencia de nucleótidos o aminoácidos.
- FASTQ: conjunto de secuencias de nucleótidos leídas junto con sus puntuaciones de calidad
- GFF/GTF: formato de anotación de características del genoma
- GenBank: formato enriquecido para secuencias de nucleótidos o aminoácidos.

I.1.1. FASTA

Guarda una o varias secuencias de nucleótidos o aminoácidos. Se pronuncia como "fast A", que significa "fast all", siendo all cualquier alfabeto. Los archivos fasta pueden tener las siguientes extensiones:

- **.fasta, .fa**: extensión FASTA genérica
- **.fna**: contiene secuencia de nucleótidos
- **.faa**: contiene secuencia de aminoácidos
- **.frn**: contiene secuencias de ARN no codificante

En la figura I.1 se puede ver la estructura de un fichero FASTA. Cada secuencia consiste en una línea descriptiva seguida de las líneas que contienen la secuencia. La línea descriptiva cuenta con un símbolo > seguido del identificativo de la secuencia. Tras un espacio, se pone la descripción de la secuencia. En algunos casos, las secuencias se encuentran en mayúscula, en minúscula o en una mezcla de ambas. Por ello, se

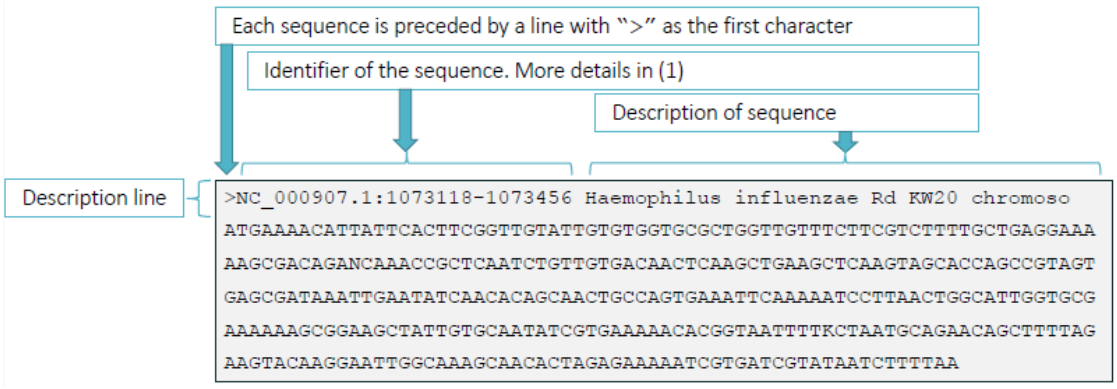


Figura I.1: *Anatomía de un fichero FASTA*

puede pasar toda la secuencia a mayúsculas o minúsculas con .upper() y .lower() respectivamente.

La secuencia se puede expandir por múltiples líneas, las cuales generalmente tienen unos 80 caracteres. No hay espacios en las líneas o líneas en blanco en una secuencia. Cada elemento de una secuencia se describe con un único carácter, y normalmente se escriben en mayúscula (aunque esto es una convención, no es estándar). No es recomendable utilizar guiones para representar gaps, ya que hay algunos programas que no los pueden manejar. Es mejor utilizar una N en el caso de secuencias de nucleótidos y X de aminoácidos (véase tabla I.1).

FASTA for genome data

A adenosine	C cytidine	G guanine
T thymidine	N A/G/C/T (any)	U uridine
K G/T (keto)	S G/C (strong)	Y T/C (pyrimidine)
M A/C (amino)	W A/T (weak)	R G/A (purine)
B G/T/C	D G/A/T	H A/C/T
V G/C/A	- gap	

FASTA for protein data

A alanine	B aspartate/asparagine	C cystidine
D aspartate	E glutamate	F phenylalanine
G glycine	H histidine	I isoleucine
K lysine	L leucine	M methionine
N asparagine	P proline	Q glutamine
R arginine	S serine	T threonine
U selenocysteine	V valine	W tryptophan
Y tyrosine	Z glutamate/glutamine	X any
* translation stop	- gap	

Tabla I.1: *Caracteres en los ficheros FASTA para datos genómicos y de proteínas.*

I.1.1.1. Parsear ficheros FASTA en Python

En informática, parsear significa leer un fichero. Se pueden leer ficheros FASTA en Python de la siguiente forma:

```
def readFasta(file):
    """ Reads all sequences of a FASTA file
        returns a dictionary """
    d = {}
    identifier = ''

    with open(file, 'r') as f:
        for linea in f:
            if linea[0] == ">": #alternativa: linea.startswith(">")
                if identifier != '':
                    d[identifier] = ''.join(d[identifier])
                descriptors = linea[1:].split()
                identifier = descriptors[0]
                #Alternativa: identifier = linea[1:linea.find(' ')]
                d[identifier] = []

            else:
                d[identifier].append(linea.strip('\n'))
        d[identifier] = ''.join(d[identifier])
    return d

readFasta("phix174/phix.fa")
```

I.1.2. FASTQ

Los ficheros FASTQ guardan secuencias de nucleótidos junto con las calidades.

Cada secuencia de un archivo FASTQ consta de 4 líneas. La primeras es la línea de descripción precedida de . Tiene un formato libre sin límite de longitud. A veces se coloca un identificador justo después de la . La segunda línea es la línea de secuencia, e incluye la secuencia propiamente dicha. Sigue normas y convenciones similares a las del fichero FASTA: normalmente en mayúsculas, sin espacios permitidos, etc. Esta línea podría estar envuelta en múltiples líneas como en un archivo FASTA. La siguiente es la línea que señala el final de la secuencia. Está marcada con el signo +. Puede ir seguida de la misma descripción dada en la primera línea. Sin embargo, es más sensato dejarla sólo con el '+' para reducir el tamaño de los ficheros. La última línea es la línea de puntuaciones de calidad: una secuencia codificada en caracteres con la calidad de las lecturas de la secuencia. Contiene un carácter por base. Los caracteres permitidos son como máximo ASCII 33-126 inclusive, que son todos imprimibles. Esta línea puede ser envuelta como en un archivo FASTA y su longitud total debe ser igual a la secuencia. Véase un ejemplo de un archivo FASTQ en la figura I.2.

Hay varias cuestiones que deben tenerse en cuenta al analizar un archivo FASTQ: Tanto el carácter como + pueden aparecer en la línea con los índices de calidad. Pueden aparecer en la primera posición, por lo que hay que tener cuidado al utilizar

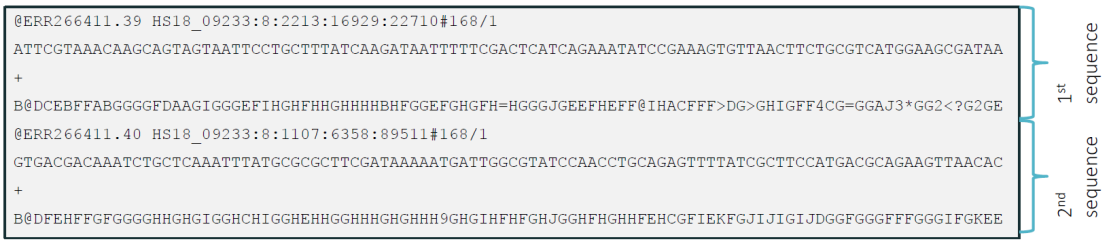


Figura I.2: *Anatomía de un fichero FASTQ*

grep o herramientas similares: por ejemplo `$> grep -v "[ +]" file.fastq` (Este comando para eliminar las líneas + y no es una buena idea). En principio, tanto la secuencia como las puntuaciones de calidad pueden tener múltiples saltos de línea. El programa de análisis debería eliminar los saltos de línea. Para reducir los problemas de análisis sintáctico, la mayoría de las herramientas producen ahora archivos fastq en los que las secuencias y las puntuaciones de calidad se dan en una sola línea, posiblemente muy larga. Por lo tanto, cada secuencia tiene exactamente 4 líneas, lo cual es conveniente.

La codificación más utilizada para las puntuaciones de calidad es el formato Sanger o codificación Phred+33. Cada carácter codifica la calidad de la lectura, Q, que se define como:

$$Q = -10 \cdot log_{10}p$$

siendo p la probabilidad de error de la lectura. Esta puntuación de calidad se utiliza porque permite una interpretación fácil de la probabilidad (Q = 10, p = 1/10; Q = 20, p = 1/100; Q = 30, p = 1000) y porque permite una codificación en un único carácter ASCII. No obstante, no se puede guardar Q directamente como un carácter porque no todos son imprimibles. Por ello, se emplea la codificación Phred+33, que codifica el valor Q como:

$$Q = ord(chr) - 33$$

donde ord() convierte el carácter a su representación ASCII. Por ejemplo, el símbolo tiene el código ASCII de 64, por lo que su Q sería de 31. Una p de 0 significa que es una lectura segura, mientras que una p de 1 indica que la lectura no es nada segura. Para calcular la p, se despeja la fórmula para que quede:

$$p = 10^{-\frac{Q}{10}}$$

Ejemplos: Para el carácter " ", el código ASCII es 126, por lo que su Q es $126 - 33 = 93$. En este caso, la p sería de $0,5 \cdot 10^{-10}$. Así, el carácter "!", cuyo ASCII es 33, tiene una Q de 0 y una p de 1.

En este caso, como las letras en mayúscula y minúscula tienen un código ASCII diferente, no se puede pasar todo a mayúsculas o minúsculas como en FASTA.

I.1.3. GFF: generic feature format

Los ficheros GFF tienen un formato de texto sin formato para representar características genómicas. Cada característica está representada por nueve columnas separadas por tabuladores:

1. seqid: id de la secuencia donde se localiza la característica. No se permiten espacios
2. source: nombre del programa o algoritmo que ha generado esta característica.
3. type: tipo de característica (region, gene, CDS, etc.)
4. start: posición donde inicia la característica, su primera base
5. end: posición donde finaliza la característica, su última base
6. score: un valor flotante que indica la confianza en la característica. Si no se conoce, se pone un punto.
7. strand: indica el sentido de la cadena, siendo + la cadena positiva y - la cadena negativa.
8. phase: puede ser 0, 1 o 2 para CDS y un punto para todo lo demás.
9. attributes: Una lista de atributos de la característica en formato tag=value separados por punto y coma (;). Se permiten espacios en blanco en estas columnas, así como tabuladores y punto y coma si se escapan. Los atributos son:
   - **ID**: El id de la característica. Debe ser único dentro del archivo gfffile. Tenga en cuenta que para las características no contiguas el id puede aparecer en varias líneas. Se considera una característica única
   - **Name**: Nombre de la característica. No es necesario que sea único.
   - **Parent**: Id del elemento padre. Indica la relación «parte de». Un elemento puede tener varios padres. En este caso, los identificadores se separan por comas.
   - **Note**: nota de texto libre
   - ...

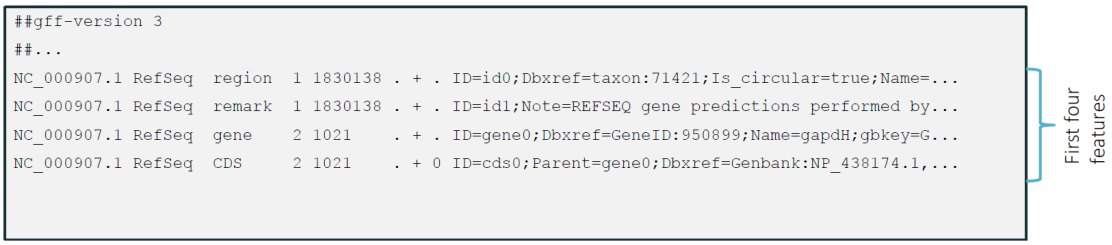
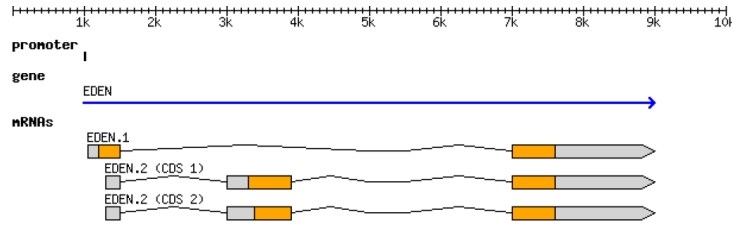


Figura I.3: *Primeras columnas de un fichero GFF.*

Hay varias cuestiones que deben tenerse en cuenta al analizar un archivo GFF: Las columnas sólo están separadas por tabuladores, pero se permiten espacios en blanco en algunos campos. Por tanto, hay que tener cuidado con grep y herramientas shell similares. Algunos caracteres se escapan en algunos campos, es decir, no queremos que los caracteres se interpreten como los caracteres que son. Esto ocurre con los puntos y coma y los espacios. Estos caracteres se escriben de otra forma; por ejemplo, un valor

de a en punto y coma debe codificarse con %3B. Para decodificar estos caracteres puede utilizar urllib. Además, una fila con triple almohadilla (###) indica que se han resuelto todas las características multilínea hasta ese punto. Hay al menos una de estas líneas al final del fichero. Las líneas que empiezan por # son comentarios. La región definida por los rasgos CDS incluye los codones de inicio y fin.



Example adapted from: https://github.com/The-Sequence-Ontology/Specifications/blob/master/gff3.md

```
1  ctg123 . gene            1000  9000  .  +  .  ID=gene00001;Name=EDEN
2  ctg123 . TF_binding_site 1000  1012  .  +  .  ID=tfbs00001;Parent=gene00001
3  ctg123 . mRNA            1050  9000  .  +  .  ID=mRNA00001;Parent=gene00001;Name=EDEN.1
4  ctg123 . mRNA            1300  9000  .  +  .  ID=mRNA00002;Parent=gene00001;Name=EDEN.2
5  ctg123 . exon            1300  1500  .  +  .  ID=exon00001;Parent=mRNA00002
6  ctg123 . exon            1050  1500  .  +  .  ID=exon00002;Parent=mRNA00001
7  ctg123 . exon            3000  3902  .  +  .  ID=exon00003;Parent=mRNA00002
8  ctg123 . exon            7000  9000  .  +  .  ID=exon00005;Parent=mRNA00001,mRNA00002
9  ctg123 . CDS             1201  1500  .  +  0  ID=cds00001;Parent=mRNA00001;Name=edenprotein.1
10 ctg123 . CDS             7000  7600  .  +  0  ID=cds00001;Parent=mRNA00001;Name=edenprotein.1
11 ctg123 . CDS             3301  3902  .  +  0  ID=cds00002;Parent=mRNA00002;Name=edenprotein.2
12 ctg123 . CDS             7000  7600  .  +  1  ID=cds00002;Parent=mRNA00002;Name=edenprotein.2
13 ctg123 . CDS             3391  3902  .  +  0  ID=cds00003;Parent=mRNA00002;Name=edenprotein.3
14 ctg123 . CDS             7000  7600  .  +  1  ID=cds00003;Parent=mRNA00002;Name=edenprotein.3
```

Figura I.4: *Ejemplo gráfico de una región genómica con el fichero GFF.*

Se puede trabajar con un fichero GFF desde la terminal de Linux. Por ejemplo, se pueden buscar todas las regiones (`grep region fichero.gff`) u obtener las terceras columnas (`cut -f3 fichero.gff`).

```python
def readGFF(file):
    """ Counts region types of a GFF file
        returns a dictionary """
    d = {}

    with open(file, 'r') as f:
        for linea in f:
            if linea[0] != "#":
                campos = linea.split("\t")
                region_type = campos[2]
                if region_type not in d: #o d.keys()
                    d[region_type] = 1
                else:
                    d[region_type] += 1
    return d

readGFF('data/haemophilus_influenzae/GCF_000027305.1_ASM2730v1_genomic.gff')
```

I.1.4. GenBank record

GenBank es una base de datos de secuencias de libre acceso mantenida por el National Center for Biotechnology Information (NCBI). Esta base de datos tiene un formato de archivo plano para representar un registro en la base de datos. Este formato codifica la secuencia, las características y las referencias de proteínas y genes.

```
LOCUS       NC_000907              1656 bp    DNA     linear   CON 02-AUG-2016
DEFINITION  Haemophilus influenzae Rd KW20 chromosome, complete genome.
ACCESSION   NC_000907 REGION: complement(36376..38031)
VERSION     NC_000907.1
DBLINK      BioProject: PRJNA57771
            Assembly: GCF_000027305.1
KEYWORDS    RefSeq.
SOURCE      Haemophilus influenzae Rd KW20
  ORGANISM  Haemophilus influenzae Rd KW20
            Bacteria; Proteobacteria; Gammaproteobacteria; Pasteurellales;
            Pasteurellaceae; Haemophilus.
```

Figura I.5: *Principio de un documento GenBank.*

Las partes de un documento GenBank (figura I.5) son:

- LOCUS: Contiene un identificador (NC_000907 en el ejemplo siguiente), la longitud de la secuencia (1656 pb), el tipo de molécula (ADN) y la fecha de la última modificación del registro.
- DEFINITION: Breve descripción de la secuencia.
- ORGANISM: Nombre científico del organismo .
- REFERENCE: Publicación científica relacionada con la secuencia. La última suele referirse al autor que envió la secuencia (esta referencia se marca con «direct submission» en lugar del título del artículo).
- FEATURES: Información sobre genes y productos génicos de la secuencia. Tiene varios subcampos:
  - Source: Campo obligatorio que describe la longitud y el organismo de la secuencia.
  - gene: región de un gen
  - CDS: región codificante de una proteína. Incluye translation, tabla, protein_id, producetname, notas, etc.

I.2. Librerías de Python

I.2.1. NumPy

NumPy (Numerical Python) es una librería de Python que permite realizar operaciones con arrays y matrices. También incluye herramientas de álgebra lineal, estadística o generación aleatoria de números. Se combina frecuentemente con otras librerías como SciPy para computaciones numéricas y pandas para análisis de datos.

I.2.1.1. Clase ndarray

Creación de arrays El elemento principal de NumPy es el objeto **ndarray** que representa arrays multidimensionales. Es capaz de realizar operaciones vectoriales eficientes (en velocidad y memoria). Al contrario que las listas de Python, todos los elementos de un ndarray son del mismo tipo. Los tipos posibles son int, float, boolean, string, etc. El atributo shape te da el tamaño del array y dtypet el tipo de los elementos.

```
import numpy as np

data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
data.shape
#Output: (3, 3)
data.dtype
#Output: dtype('int64')
```

Se puede crear un array a partir de cualquier otra secuencia (listas u otros ndarrays), utilizando la función numpy.array. Es posible especificar en la construcción el tipo del array (dtype) y la dimensión. Las dimensiones son importantes para cuando haya que fusionar arrays, ya que deberán coincidir. Además, la dimensión debe coincidir en todo el array, es decir, debe ser homogéneo.

```
x = np.array([1, 2, 3], ndmin = 2)
# array([[1, 2, 3]])
y = np.array([1, 2], dtype = 'int32')

error = np.array([1, 2, 3], [4, 5])
```

Otras funciones para crear e inicializar arrays/matrices son:

- Matrices de ceros: numpy.zeros y numpy.zeros_like (el primero crea un array de las dimensiones dadas, el segundo crea un array como el que se le pasa, es decir, con sus dimensiones y el tipo).
- Matrices de unos: numpy.ones y numpy.ones_like
- Matriz identidad: numpy.eye y numpy.identity
- La función numpy.arange se comporta de forma similar a la función estándar de python range, pero devuelve un ndarray.
- La función numpy.linspace devuelve números espaciados uniformemente en un intervalo especificado. Se proporciona el primer y último valor y el número total de valores. La función numpy.logspace hace algo similar pero de forma logarítmica.

```
np.zeros((3,2))
#array([[0., 0.], [0., 0.], [0., 0.]])

x = np.array([1, 2, 3])
np.ones_like(x)
#array([1, 1, 1])
```

```
y = np.arange(10)
#[0 1 2 3 4 5 6 7 8 9]

np.linspace(-np.pi,np.pi,10)
#array([-3.14159265, -2.44346095, -1.74532925, -1.04719755, -0.34906585,
    0.34906585, 1.04719755, 1.74532925, 2.44346095, 3.14159265])
```

Operaciones básicas con arrays Las operaciones aritméticas pueden realizarse con ndarrays utilizando la misma sintaxis que la empleada para los escalares y evitando el uso de bucles explícitos.

```
data = np.array([[1, 2, 3], [4, 5, 6]])
data + 1
#array([[2, 3, 4], [5, 6, 7]])
data * 10
array([[10, 20, 30], [40, 50, 60]])
```

Arrays del mismo tamaño se pueden sumar y multiplicar celda a celda (si son de distinto tamaño no funciona):

```
data + data
#array([[ 2, 4, 6], [ 8, 10, 12]])
data * data
#array([[ 1, 4, 9], [16, 25, 36]])
```

Todas las operaciones booleanas (>, <, ==, etc) se aplican elemento a elemento y se devuelve un array de booleanos:

```
data = np.array([[1, 2, 3], [4, 5, 6], [4, 5, 6]])
#array[[1 2 3], [4 5 6], [4 5 6]]
data > 4
#array([[False, False, False], [False, True, True], [False, True, True]])
```

Ejercicios Construye una matriz de 5x5 elementos en la que los elementos de la diagonal sean 10 y los elementos fuera de la diagonal sean 5.

```
np.identity(5)*5+5
(np.identity(5)+1)*5

def diag2(n, f):
   ''' f: factor, n: dimensiones del array'''
   return (np.identity(n) + 1) * f
diag2(5,5)
```

Crea un array que contenga los días de la semana. ¿Cuál es su dtype?

```
weekdays_matrix = np.array(['Monday', 'Tuesday', 'Wednesday',
    'Thursday', 'Friday', 'Saturday', 'Sunday'])
print(weekdays_matrix.dtype)
#<U9
#El 9 es la longitud de palabra más grande del array
```

I.2.1.2. Indexación y slicing

Para arrays unidimensionales, el funcionamiento es como en el caso de las listas:

```
x = np.arange(10)
#array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
x[4]
#4
x[2:7]
#array([2, 3, 4, 5, 6])
x[-3:]
#array([7, 8, 9])
```

Si se asigna un valor a un slice, el valor se asigna a todas las posiciones en el slice.

```
x[4:7] = -1
#array([ 0, 1, 2, 3, -1, -1, -1, 7, 8, 9])
x = np.arange(10)
#array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
x[0::2] = [19,17,15,13,11]
#array([19, 1, 17, 3, 15, 5, 13, 7, 11, 9])
```

Ten en cuenta que si obtienes una referencia a un slice, los elementos no se copian y sólo se mantiene la referencia a los elementos del slice. Esto significa que si después utilizas el slice para modificar sus valores, éstos se modifican en el array original. Si queremos una copia del slice y no una referencia a los valores en los arrays originales, se puede utilizar el método copy().

```
x = np.ones((5,3), dtype=int)
#array([[1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1]])
y = x[:5:2,1:] #aquí se seleccionan primero filas y luego columnas
#array([[1, 1], [1, 1], [1, 1]])
y[:] = 25
print(x)
#array([[ 1, 25, 25], [ 1, 1, 1], [ 1, 25, 25], [ 1, 1, 1], [ 1, 25,
    25]])

x = np.ones((5,3), dtype=int)
z = x[:3].copy()
#array([0, 1, 2])
y[2] = 25
print(x)
#[0 1 2 3 4 5 6 7 8 9]
print(y)
```

```
#[ 0  1 25]
```

En arrays multidimensionales, se utilizan comas para separar los slices para cada dimensión.

```
data = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14,
    15, 16]])
#array([[ 1, 2, 3, 4],
# [ 5, 6, 7, 8],
#[ 9, 10, 11, 12],
#[13, 14, 15, 16]])
data[1,1:3]
#array([6, 7])
data[1:3]
#array([[ 5, 6, 7, 8],
# [ 9, 10, 11, 12]])
data[:, 1:3]
#array([[ 2, 3],
# [ 6, 7],
# [10, 11],
# [14, 15]])
data[1:4:2, 1:4:2]
#array([[ 6, 8],
# [14, 16]])
```

Indexación booleana Es posible utilizar un array de valores booleanos (Verdadero/Falso) para seleccionar elementos del ndarray.

```
data = np.random.randn(10, 3) #generación de matriz 10x3 con números de
    distribución normal
data[data>0] #números positivos
ix = data[:,0] > 0 #índices booleanos para las filas en las que el
    primer elemento es positivo
#array([ True, True, False, False, True, False, False, False, False,
    True])
selected = data[ix, :] #seleccionar las filas que empiezan por un número
    positivo
```

La indexación booleana siempre crea una copia, a diferencia del slicing. Es posible combinar varias operaciones booleanas utilizando & (and), | (or) y $\sim$ (negación). Además, se puede utilizar para asignar valores a los elementos de un array que cumplan con una condición.

```
data[(data[:, 0] > 0) & (data[:, 2] < 0), :] #selecciona las filas en
    las que el primer elemento es positivo y el último negativo
data[data < 0] = 0 #asigna el valor de 0 a todos los elementos negativos
```

Indexación fancy También es posible indexar utilizando matrices de enteros de forma similar al slicing. Estos enteros indican los índices de columna o fila a seleccionar. Este tipo de indexación es útil, por ejemplo, para seleccionar filas y columnas en un orden distinto al original.

```
nrows = 5
ncols = 3
p = 1
data = np.zeros((nrows, ncols))
for i in range(ncols):
    data[:, i] = np.arange(nrows)*p
    p*=10
#array([[ 0.,  0.,  0.],
#       [ 1., 10., 100.],
#       [ 2., 20., 200.],
#       [ 3., 30., 300.],
#       [ 4., 40., 400.]])
data[[3, 1]] #selección de las filas 3 y 1 en ese orden
#array([[ 3., 30., 300.],
#       [ 1., 10., 100.]])
data[:, [2, 0]]
#array([[ 0.,  0.],
#       [100.,  1.],
#       [200.,  2.],
#       [300.,  3.],
#       [400.,  4.]])
```

Ejercicios En el siguiente código, ¿qué diferencia supondría `y = 100 * y` con `y[:] = 100*y`?

```
x = np.array([1, 2, 3, 4, 5])
y = x[:3]
y = 100*y
print(y) # [100 200 300]
print(x) # [1 2 3 4 5]
y = x[:3]
y[:] = 100*y
print(y) # [100 200 300]
print(x) # [100 200 300 4  5]
```

De ambas formas, el resultado para y es igual; la diferencia se encuentra en el array original x. En la primera variante, x no se ve modificado, mientras que en la segunda variante, los valores de x sí se han visto modificados.

El siguiente array contiene una lista de 100 notas aleatorias de 0 a 10. Crea un array que tenga una nota cualitativa para cada nota numérica:

```
n = 100
np.random.seed(13)
notas = np.random.rand(n)*10
```

```
notas_cual = np.array(notas, dtype=str)
notas_cual[notas < 5] = "SUSPENSO"
notas_cual[(notas >= 5) & (notas < 7) ] = "APROBADO"
notas_cual[(notas >= 7) & (notas < 9) ] = "NOTABLE"
notas_cual[notas >= 9] = "SOBRESALIENTE"
```

Ahora cuenta el número de suspensos, aprobados, notables y sobresalientes.

```
np.sum(notas_cual == "SUSPENSO")
np.sum(notas_cual == "APROBADO")
np.sum(notas_cual == "NOTABLE")
np.sum(notas_cual == "SOBRESALIENTE")
```

Hay más funciones de Numpy que no hemos visto. Por ejemplo, `np.unique` proporciona todos los valores únicos. Así, el ejercicio anterior se podría resolver de la siguiente forma:

```
for value in np.unique(notas_cual):
    print("# de ", value, ":", np.sum(notas_cual == value))
```

I.2.1.3. Operaciones con arrays

Transponer La función `transpose` o `T` permite intercambiar las filas por columnas de un array.

```
data = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14,
    15, 16]])
data.T # o data.transpose()
```

También es posible transponer arrays multidimensionales especificando los índices de los ejes que se quieren intercambiar.

Cambiar la forma de un array (reshape y ravel) El método `ndarray.reshape()` cambia la forma de un array sin cambiar su contenido, solo lo reajusta. El número de celdas totales en ambos arrays debería ser igual. El método `ndarray.ravel()` comprime un array en uno de una sola dimensión. Además, el método `np.concatenate` concatena dos arrays a lo largo de un eje, dado que el tamaño de la otra dimensión sea igual. Algunos ejemplos son:

```
x = np.arange(1, 13)
print(x)
y = x.reshape(3, 4) #cambia a dimensiones 3x4
print(y)
z = y.ravel() #vuelve a 1 dimensión
print(z)

x = np.zeros((3, 4))
```

```
print(x)
y = np.ones((3, 2))
print(y)
np.concatenate((x, y), axis = 1) #el eje 1 son las columnas
np.concatenate((x[:, :2], y), axis = 0) #eje 0 son las filas, y sólo se
    obtienen las primeras dos columnas de x
```

I.2.2. Matplotlib

Matplotlib es una librería de gráficos. Se suele importar como `import matplotlib.pyplot as plt`. La función más sencilla es `plot`.

```
x = np.linspace(0.1, 4, 40) #[0.1, 0.2, 0.3, 0.4, 0.5, ...]
y = 1/x #[10., 5., 3.3333, 2.5, 2., ...]
plt.plot(x, y)
```

También se pueden imprimir varias líneas a la vez:

```
x = np.linspace(-np.pi, np.pi, 40)
y1 = np.sin(x)
y2 = np.cos(x)
# Use dummy variable _ to get the return value of plot to avoid printing
    its reference
_ = plt.plot(x, y1, x, y2)
```

Opciones de trazado Se puede especificar la forma de trazado al proporcionar una cadena que incluya un identificador.

```
x = np.arange(0.1,4,0.1)
y = 1/x
plt.plot(x, y, 'm*')
```

En el caso anterior, m representa el color magenta y el asterisco la forma de cada punto en el gráfico. Así, las líneas discontinuas se muestran como dos guiones, guion punto alterna ambos símbolos, la k representa el color negro, etc. Todas las opciones se pueden encontrar en la guía.

También se pueden retocar otras propiedades:

- linewidth: el ancho de la línea en píxeles
- markeredgecolor: el color del borde del marcador
- markerfacecolor: el color interno del marcador
- markerzie: el tamaño del marcador

En general, estos parámetros aplican a todo el gráfico. Sin embargo, se puede establecer anchos y colores distintos para cada una de las líneas llamando al plot varias veces:

```
x  = np.arange(0.1,20,0.5)
y  = np.exp(0.1*x)
y1 = y * np.sin(x)
y2 = y * np.cos(x)

_ = plt.plot (x, y1, ':ms', linewidth=2, markeredgecolor='k',
    markerfacecolor='r', markersize=5)
_ = plt.plot (x, y2, '-co', linewidth=11, markeredgecolor='b',
    markerfacecolor='c', markersize=11)
```

Tipos de gráficos Se pueden crear distintos tipos de gráficos:

- Gráfico de tarta (pie plot): mediante `plt.pie(x, Explode)`, siendo Explode la distancia a la que se debe separar la fracción.
- Histograma: `hist(x, nbins)`. Por defecto, se crean 10 cajas o bins equidistantes. Si x es una matriz, entonces se crea un histograma por columna.
- Diagrama de barras (bar plot): la función `bar(x, y, width, bottom, style...` crea un gráfico de barras verticales u horizontales (`barh`). El parámetro width indica la separación entre las barras, y bottom permite poner barras apiladas.
- Contour: Esta función `contour(Z, n)` traza isolíneas desde la matriz Z, donde Z se interpreta como alturas con respecto al plano xy. Z debe ser al menos una matriz de 2x2. El número de niveles puede especificarse utilizando el parámetro «n»; de lo contrario, se calculan automáticamente a partir de las alturas máxima y mínima de Z.
- Pcolor: La función `pcolor(c)` dibuja una matriz coloreada con colores dados por C. El gráfico muestra una columna y una fila menos que las columnas y filas de la matriz. Esta función (si no se dan otros parámetros) obtiene el color para la celda (i, j) del elemento (i, j) de la matriz C usando el mapa de colores actual.
- Diagrama de dispersión (scatter plot): La función `scatter(x, y, s = 20, c=u'b', ...` crea un diagrama de dispersión con coordenadas x e y. Los vectores x e y deben tener el mismo tamaño. El tamaño de los puntos puede especificarse con s y su color con c. Tanto el tamaño como el color pueden darse para todos los puntos juntos utilizando un único valor o para cada punto individual utilizando un vector. Para la primera configuración, basta con dar un único escalar para el tamaño y una cadena para el color. Para la segunda, se debe dar un vector del mismo tamaño de x e y para los tamaños. Para los colores, se puede especificar un vector que contenga indixes f el mapa de colores, o un array de tamaño len(x) x 3 con colores en RGB.

Opciones globales para gráficos Todos los gráficos admiten una serie de funciones que permiten crear un gráfico "decente":

- figure: crea una figura del tamaño especificado. En esta opción también se puede especificar el color de fondo.
- subplot(nrows, ncols): crea una matriz de gráficos de filas nrows y columnas ncols. El último parámetro indica sobre qué gráfico vamos a trabajar. También hay más parámetros como sharex o sharey que permite crear ejes compartidos entre todos los subgráficos. También se debe indicar mediante índice el subgráfico que se quiere modificar a continuación.
- xlabel ylabel: permite indicar el nombre de los ejes x e y.
- title: permite especificar el título del gráfico.
- legend: añade la leyenda al gráfico. Se debe llamar esta opción una vez se haya ejecutado la función de creación del gráfico.
- colorbar: incluye una barra de colores de referencia.
- xlim ylim: modifica los límites de los ejes que se muestren en las gráficas.
- xticks yticks: modifica las marcas de los ejes.

Un ejemplo de como aplicar todo esto sería el siguiente (el resultado se ve en la figura I.6):

```
plt.figure(num=None, figsize=(12, 4), facecolor='c')

# Data
x = np.linspace(-np.pi,np.pi,20)
Y1 = np.sin(x)
Y2 = np.cos(x)

# Graph using legend and labels in the axis
plt.subplot(1,2,1) # Create a graph with two subplots. We start working
    on the first
plt.xlabel('x')
plt.ylabel('y')
plt.plot(x, Y1, label='sen')
plt.plot(x, Y2, label='cos')
plt.legend(loc=2) # Called after finishing the plot

# Graph with x label, title, with the limits of the axis modified
# and also the labels for the x axis
plt.subplot(1,2,2) # Now we use the second subplot
plt.xlabel('x')
plt.title('Seno y coseno')
plt.xlim([-3,3])
plt.xticks(range(-3,4), ['a','b','c','d','e','f','g'])
plt.plot(x, Y1, label='sen')
_ = plt.plot(x, Y2, label='cos')
```

Otro ejemplo sería:

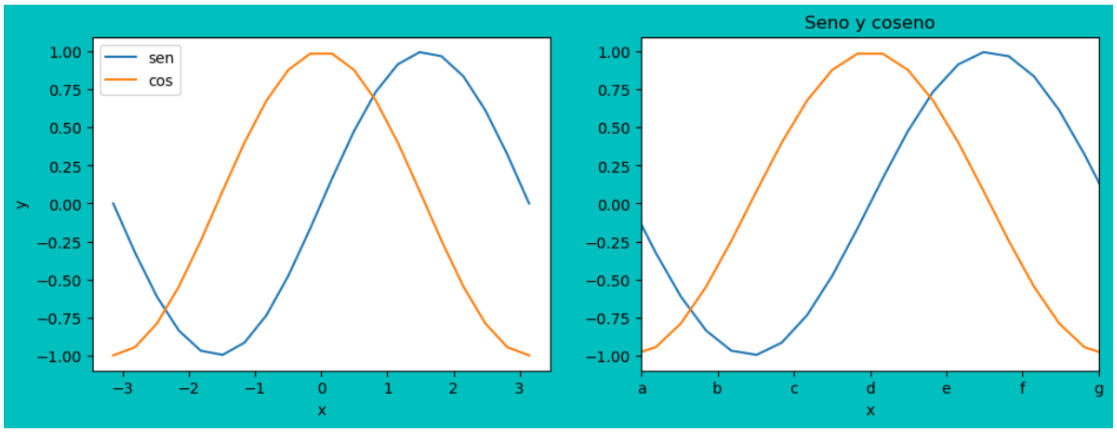


Figura I.6: *Gráfico de ejemplo*

```
def sigmoid(V, V0, W=12):
    """ Sigmoidal function

    Parameters
    ----------
    V: float or array of floats
        Membrane potential
    V0: float
        Position of the sigmoid function
    W: float
        Width of the transition region

    Returns
    -------
    y: float or array of floats
        Sigmoidal function
    """
    return 1/(1+np.exp(-(V-V0)/W))

V = np.arange(-100, 100, 1) #if we use linspace, instead of the steps we
    put how many values we want
y = sigmoid(V, -40, 12)

fig, axs = plt.subplots(1, 2)
#varying V0
ax = axs[0]
for V0 in [-40, 0, 40]:
    y = sigmoid(V, V0, 12)
    ax.plot(V, y, label = f'{V0:.0f}mV')
ax.set_xlabel('V [mV]')
ax.legend(title = '$V_0$')

#Varying W
ax = axs[1]
for w in [1, 10, 20]:
```

```
    y = sigmoid(V, 0, w)
    ax.plot(V, y, label = f'{w:.0f}mV')
ax.set_xlabel('V [mV]')
ax.legend(title = '$w$')
plt.show()
```

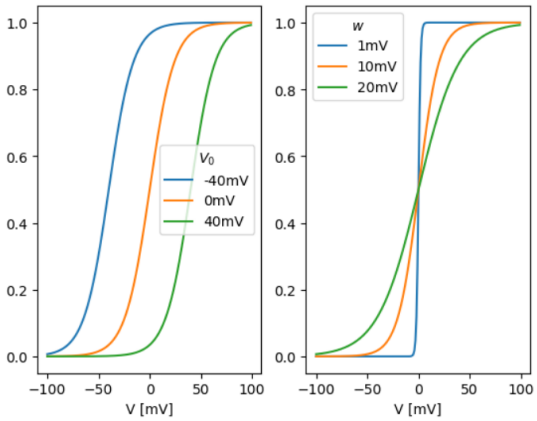


Figura I.7: *Otro ejemplo*

I.3. Biopython

Biopython es una librería Python con herramientas para computación biológica. Se trata de un proyecto de código abierto miembro de la Open Bioinformatics (OBF). Una fundación que promueve el código abierto en la investigación biológica. El objetivo de esta librería es facilitar al máximo el uso de python en bioinformática para el análisis sintáctico de archivos, funciones para trabajar con secuencias y características, trabajar con sitios bioinformáticos online como NCBI o ExPASy, alineamientos, etc.

I.3.1. Seq

La clase Seq es la clase básica para representar secuencias en Biopython. Se representan las secuencias biológicas como cadenas. Así, se puede iterar sobre las bases, realizar slicing, concatenar secuencias y utilizar otros métodos de cadenas. Todas estas operaciones devuelven nuevos objetos de tipo Seq, que son inmutables, como los strings.

Esta clase tiene métodos especiales de ADN como `complement` o `reverse_complement`.

```
dna = Seq('GATTACA')
print(dna.complement()) #CTAATGT
print(dna.reverse_complement()) #TGTAATC
```

También permite la transcripción mediante `transcribe` y `back_transcribe` para obtener el ARN y la traducción para obtener la secuencia de aminoácidos (`translate`). Se puede especificar la tabla de traducción y si se desea parar justo antes del primer codón de stop.

```
cds = Seq('GTGTTTTTGGTGTGGTGA')
mRNA = cds.transcribe()
print(mRNA) #GUGUUUUUGGUGUGGUGA
print(mRNA.back_transcribe()) #GTGTTTTTGGTGTGGTGA

mRNA = Seq('UUGUUUUUGGUGUGGUGA')

print("Translate using standard table: ", mRNA.translate()) #Translate
    using standard table: LFLVW*
print("Translate until stop codon : ", mRNA.translate(to_stop=True))
    #Translate until stop codon : LFLVW
print("Using a different table    : ", mRNA.translate(table=2)) #Using a
    different table :  LFLVWW
print("by name                    : ", mRNA.translate(table="Vertebrate
    Mitochondrial")) #by name               :  LFLVWW
# This next line raises an error because the sequence is not a CDS for
    table=2
#print("CDS-like RNA               : ", mRNA.translate(table=2,
    cds=True)) #CDS-like RNA           :  MFLVW
# But it is a CDS-like sequence for table=11 (Bacterial)
print("CDS-like RNA               : ", mRNA.translate(table=11,
    cds=True)) #CDS-like RNA           :  MFLVW

cds = Seq('GTGTTTTTGGTGTGGTGA')
prot = cds.translate(table=11, cds=True)
prot = cds.translate()
prot #Seq('VFLVW*')
```

Si la secuencia es un CDS, debe tener un codón de inicio y final; si no, se produce un error. No es necesario seguir el flujo natural: se pueden obtener los aminoácidos directamente desde el ADN sin tener que pasar previamente por el ARN.

I.3.1.1. Tablas de traducción

Se pueden obtener las tablas de traducción:

```
from Bio.Data import CodonTable

bacterial_table = CodonTable.unambiguous_dna_by_name['Bacterial']
print(bacterial_table)
```

I.3.2. SeqRecord

SeqRecord consta de una secuencia (objeto Seq) junto con id, descripción, anotaciones, características, etc. Se puede crear un SeqRecord a partir de un objeto Seq, de un archivo fasta, genbank, etc.

```
from Bio.SeqRecord import SeqRecord
```

```
sr = SeqRecord(Seq('AAA'), id='1', description='Simple seq',
    annotations={"molecule_type": "DNA"})
print(sr)
```

I.3.3. SeqIO

IO viene de Input Output. Bio.SeqIO proporciona acceso a la mayoría de los formatos de archivos bioinformáticos. La función `read()` carga un único archivo de secuencia. Obtiene como parámetros el nombre y formato del archivo. Si hay más de un registro esta función genera un error. Esta función devuelve un SeqRecord.

```
from Bio import SeqIO

record = SeqIO.read("phix174/phix.fa", "fasta")
print(record)
```

La función `parse` recupera varias secuencias de un archivo. Los parámetros son los mismos que read(): Nombre y formato del fichero. Esta función devuelve un iterador que permite recorrer todas las secuencias y puede iterar con un bucle

```
for record in SeqIO.parse("other/ls_orchid.fasta", "fasta"):
    print(record)
    print()
```

Otra función es `to_dict`, la cual crea un diccionario del iterador con todas las secuencias del fichero. El problema es que se guarda en la memoria, por lo que puede ralentizar el ordenador en caso de ficheros grandes.

```
iterator = SeqIO.parse("other/ls_orchid.gbk", "genbank")
records_dict = SeqIO.to_dict(iterator)
print(records_dict['Z78533.1'])
```

La función `index()` crea un pseudo diccionario a partir de un fichero. Admite como parámetros el nombre y el formato del archivo. Esta función no almacena todo en memoria, sino que obtiene la información del disco cuando es necesario. Almacena la posición de cada secuencia en el fichero para un acceso rápido. Desde el punto de vista del usuario es como un diccionario de secuencias. Es importante cerrar el fichero cuando ya no se utilice.

```
records_dict = SeqIO.index("other/ls_orchid.gbk", "genbank")
print(records_dict['Z78533.1'])
records_dict.close()
```

También se puede trabajar con ficheros comprimidos para ahorrar espacio en el disco:

```
import gzip
```

```
with gzip.open("arabidopsis_thaliana/GCF_000001735.3_TAIR10_rna.fna.gz",
    "rt") as f:
    total_len = 0
    for sr in SeqIO.parse(f, "fasta"):
        total_len += len(sr.seq)
    print(total_len)
```

Trabajar con índices y archivos comprimidos es más complicado, ya que la posición de cada secuencia no es fácil de obtener. El formato Blocked GNU Zip Format ofrece una buena solución. Comprime los datos por bloques para permitir la indexación. La función Index() puede trabajar con el formato de archivo BGZF.

I.3.4. Atributos de SeqRecord

Los atributos más importantes de SeqRecord son:

- id y nombre: identificador y nombre de la secuencia
- seq: objeto Seq que contiene la secuencia actual
- annotations: diccionario de Python con toda la información de la secuencia. Se pueden encontrar tipos de moléculas, topología, fecha, accessiones, versión de secuencia, gi, palabras clave, fuente, organismo, taxonomía, referencias, etc.
- features: lista de un objeto SeqFeature que contiene la información sobre las distintas características encontrada en una secuencia.

```
records_dict = SeqIO.index("other/ls_orchid.gbk", "genbank")
record = records_dict['Z78533.1']
records_dict.close()

print("ID:",record.id, "Name:", record.name)
print("Description:",record.description)
print(record.annotations)
print(record.features)
```

I.3.4.1. SeqFeature: características de SeqRecord

SeqFeature es una clase que incluye información sobre las características de una secuencia. Sus atributos principales son:

- type: descripción de la característica. Puede ser "gene", "CDS", "region", etc. Es similar al campo 3 de un fichero gff.
- location: región donde se encuentra la característica. Contiene las posiciones de inicio y final. Este atributo también puede manejar características localizadas en múltiples regiones.

- qualifiers: un diccionario de Python con información sobre la característica.

```
from Bio import SeqFeature

records_dict = SeqIO.index("other/ls_orchid.gbk", "genbank")
print(len(records_dict['Z78533.1'].features), "features found")

for feature in records_dict['Z78533.1'].features:
    print("----------------------------------------")
    print(feature)

records_dict.close()
```

Esta información está incluida en SeqRecord cuando se lee un fichero que contenga la información, como es el caso de GenBank. Por ejemplo, si se abre un fichero FASTA, SeqRecord no encontrará ninguna característica.

Operador in El operador `in` se utiliza para comprobar si una posición concreta se encuentra en una característica. Por ejemplo, el siguiente código imprime la característica que incluye la posición 10000.

```
record =
    SeqIO.read('haemophilus_influenzae/GCF_000027305.1_ASM2730v1_genomic.gbff',
    'genbank')

position = 10000
for feature in record.features:
    if position in feature:
        print(feature)
        print('-----------------------')
```

Este operador también funciona con objetos Location.

I.3.4.2. SeqFeature: método extract

El método `extract()` extrae las características de la secuencia completa. El siguiente código itera sobre las secuencias de un fichero y después sobre las características de cada secuencia, extrayendo las características CDS y generando la proteína asociada utilizando la tabla de traducción correspondiente.

```
n_cds = 0
file_iter =
    SeqIO.parse('haemophilus_influenzae/GCF_000027305.1_ASM2730v1_genomic.gbff',
    'genbank')
#file_iter =
    SeqIO.parse('plamodium_falciparum/GCF_000002765.3_ASM276v1_genomic.gbff',
    'genbank')

for seqrec in file_iter:
```

```
    for feature in seqrec.features:
        if feature.type == 'CDS':
            n_cds += 1
            mRNA = feature.extract(seqrec)
            try:
                transl_table = 1
                if 'transl_table' in feature.qualifiers:
                    transl_table =
                        int(feature.qualifiers['transl_table'][0])
                p = mRNA.seq.translate(table=transl_table, cds=True)
            except:
                print("Protein {0} in gene {1} could not be translated!".
                      format(feature.qualifiers['protein_id'][0],
                          seqrec.id))

print(n_cds)
```

I.3.4.3. SeqRecord: slicing

Es posible utilizar slicing con SeqRecord. El SeqRecord que se devuelve contiene solo las características que se incluyen en la subsecuencia. Las anotaciones no se copian en las subsecuencias.

```
record =
    SeqIO.read('haemophilus_influenzae/GCF_000027305.1_ASM2730v1_genomic.gbff',
    'genbank')
print(record.id)
print("# of feature: ", len(record.features))
print("# of annotations: ", len(record.annotations))
print(record.features[1200])

sub_record = record[612623:620000]
print(sub_record.id)
print("# of feature: ", len(sub_record.features))
print("# of annotations: ", len(sub_record.annotations))

print(sub_record.features[0])
```

Capítulo II

Acceso programático a bases de datos biomédicas

II.1. Introducción

En este bloque queremos acceder mediante los programas a las bases de datos biomédicas y poder interactuar con ellas. Para las bases de datos biomédicas hay que hacer peticiones desde el protocolo HTTP. Un protocolo es una forma de interacción entre dos partes: queremos hacer una petición a la base de datos a partir de una entrada para que nos devuelva una respuesta que podamos utilizar posteriormente. Tenemos que usar los protocolos intermediarios para poder interactuar con la base de datos. Como esto puede ser un lío, se diseñaron las API REST. Un API es application programming interface, es decir, un protocolo o una manera de interactuar con otra parte en un formato concreto. REST tiene que ver con el protocolo HTTP de base. El protocolo HTTP se utiliza para navegar por internet. Muchas aplicaciones exponen un API REST, lo que quiere decir que hay unas ciertas órdenes o funciones que se pueden llamar para interactuar con la aplicación. Hay una serie de endpoints, que son cada una de las funciones que están disponibles para llamar: un endpoint para buscar un organismo concreto, para descargar una cierta secuencia, etc.

Hay varias maneras de acceder a bases de datos en línea (nivel de abstracción o automatización ascendente):

- **Copiar la base de datos de manera local**: esto es un proceso manual que hace que la base de datos se quede rápidamente desactualizada. Se puede descargar un fichero de un servicio web en Linux mediante `wget url/base/de/datos/fichero.txt`
- **Rellenar formularios**: todavía el sistema no ha dado acceso al público a sus servicios mediante API, pero se puede escribir en Python un programa que rellene el formulario como si fuera un humano para acceder a los datos cambiando los valores de los parámetros. En general, la sintaxis de una URL es la siguiente:

  scheme:[//host]path[?query][#fragment]

  siendo scheme HTTP o HTTPS cuando el protocolo está cifrado y es seguro (la s viene de secure), host el servidor que contiene la base de datos a la

que se quiere acceder, el path la ruta para llegar a una parte del servidor (como si fueran carpetas) y la interrogración el marcador de los argumentos (las queries se escriben en formato nombre=valor&nombre2=valor2). Así, lo que queremos hacer con Python es codificar los distintos argumentos para poder acceder a varias entradas de forma automática. Por ejemplo, en el formulario tipo https://www.ebi.ac.uk/Tools/dbfetch/dbfetch?db=ena_sequence&id=J00231&style=raw, en lugar de rellenar el formulario varias veces, queremos cambiar el valor de los distintos argumentos (id=J00231, id=J00232, id=J00233, db=afdb, etc.).

- **Peticiones HTTP directas:** esto se puede conseguir mediante la librería `requests` en Python o mediante la terminal. Se emplean las URLs de los distintos servidores con métodos como GET y POST.
- **Uso específico de API REST:** Hay algunos servicios que proporcionan APIs de los servidores para poder utilizarlos desde Python. Esto se conoce como APIs de alto nivel o SDKs. Esta es la mejor manera, ya que es más cómoda y más fácil de utilizar, pero no está disponible en todos los sistemas. Estas API REST están muy encapsuladas, por lo que no disponen de URLs ni GET o POST.

II.1.1. Acceso programático en formularios

Hay varias librerías disponibles en Python para automatizar el acceso y procesado de las URL:

- `urllib`: acceso de bajo nivel, más adecuado para programación de red.
- `requests`

La librería requests tiene el comando get, que es el método HTTP más común.

```
import requests
url =
    "https://www.ebi.ac.uk/Tools/dbfetch/dbfetch?db=ena_sequence&id=J00231&style=raw"
response = requests.get(url)
print(response.text)
```

En el método get de requests se puede utilizar el argumento `params` para pasar los distintos argumentos. Este argumento debe ser un diccionario con los distintos parámetros:

```
import requests

ebi_url = 'https://www.ebi.ac.uk/Tools/dbfetch/dbfetch'

response = requests.get(ebi_url,
    params={'db': 'ena_sequence', 'id':'J00231' , 'style':'raw'})

print(response.text)
```

II.2. JSON

Hay distintas formas de poder representar un fichero:

- Texto plano
- HTML (HyperText Markup Language)
- XML (eXtensible Markup Language)
- JSON (JavaScript Object Notation)

La notación JSON es similar a la de los diccionarios en Python conceptualmente: las claves son strings y los valores pueden ser strings, números, listas o incluso diccionarios. Técnicamente es una forma de estructurar la información (figura II.1) en formato ASCII.

Figura II.1: *Ejemplo de un fichero JSON que describe a una persona. Las llaves representan estructuras y los corchetes listas.*

Python cuenta con una librería llamada `json`.

```
import json

json_data = '{"name":"John", "age":30, "city":"New York"}'

print(json_data)
```

Para cargar un JSON, se utiliza el método `json.loads()`. Así, el JSON se guarda en un diccionario que posteriormente se puede guardar serializado en disco. Otra función es `json.dumps()`, el cual coge un diccionario en una cadena tipo JSON. La función `json.dump` escribe un objeto JSON en un fichero.

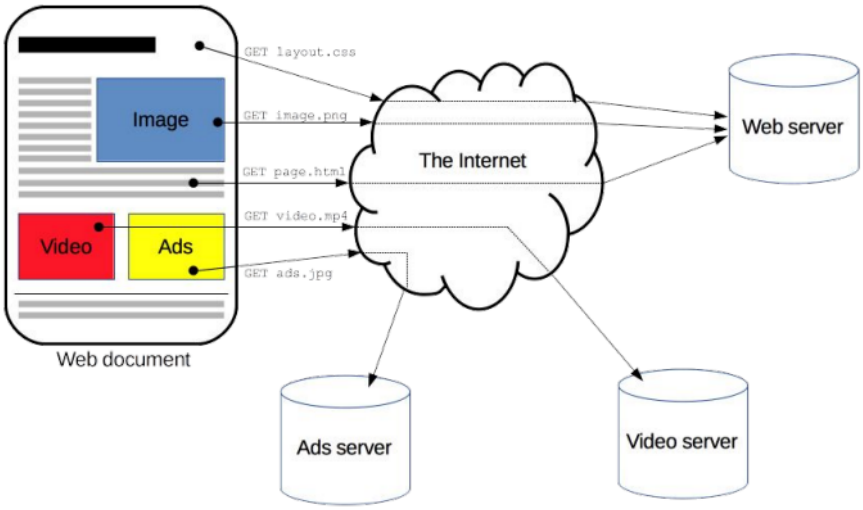
dict = json.loads(str)	Convierte un string en un diccionario
str = json.dumps(dict)	Convierte un diccionario en un string
json.dump(dict, file)	Escribe un objeto JSON (dict) en un fichero)

Tabla II.1: *Resumen*

II.3. Protocolo HTTP

Los protocolos de acceso a las bases de datos se han montado sobre el protocolo HTTP. HTTP es un protocolo para obtener recursos en la web, como documentos HTML, imágenes y otros. Es un **protocolo cliente-servidor**, lo que significa que las peticiones las inicia el destinatario, normalmente el navegador web.

Una página web es una colección de recursos: imágenes, vídeos, anuncios, etc. Estos elementos pueden estar en sitios distintos, y cuando la página se carga, se embebe en el código y se muestra.



Los clientes y los servidores se comunican intercambiando mensajes individuales (a diferencia de un flujo de datos). Los mensajes enviados por el cliente, normalmente un navegador web, se denominan **peticiones (requests)** y los mensajes enviados por el servidor como respuesta se denominan **respuestas (responses)**.

II.3.1. Aspectos básicos de HTTP

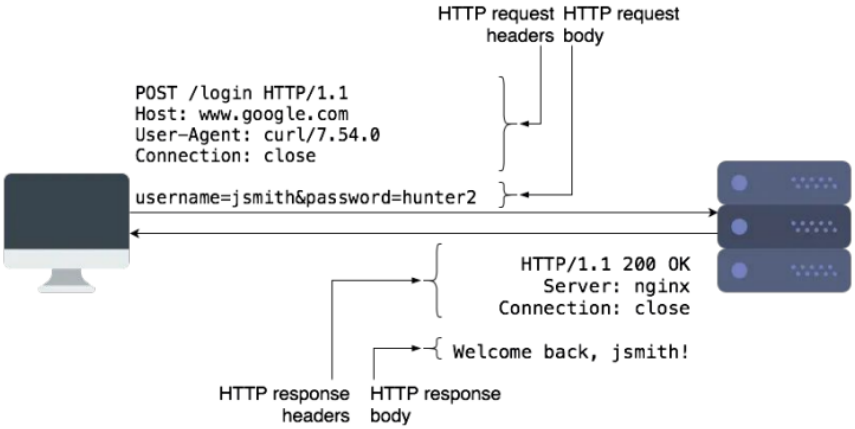
HTTP se diseñó para ser sencillo y legible para los humanos. Los mensajes HTTP pueden ser leídos y comprendidos por humanos, lo que facilita las pruebas a los desarrolladores y reduce la complejidad para los recién llegados. Además, HTTP es extensible: Las cabeceras HTTP hacen que este protocolo sea fácil de ampliar y experimentar. Se pueden introducir nuevas funciones mediante un simple acuerdo entre un cliente y un servidor sobre la semántica de una nueva cabecera.

HTTP no tiene estado, es decir, no existe ningún vínculo entre dos solicitudes que se realizan sucesivamente en la misma conexión. Las cookies HTTP permiten el uso de sesiones con estado (por ejemplo, para utilizar cestas de la compra de comercio electrónico).

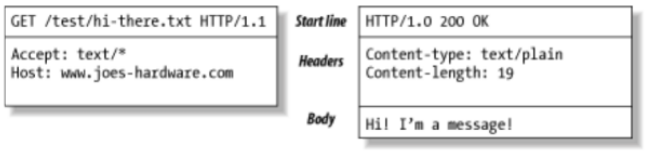
II.3.2. Flujo HTTP

Cuando un cliente quiere comunicarse con un servidor realiza los siguientes pasos:

1. Abre una conexión TCP: la conexión se utiliza para enviar una petición, o varias, y recibir una respuesta.
2. Envía un mensaje HTTP
3. Espera una respuesta



Las peticiones son puro texto (no hay datos binarios). La estructura es la siguiente: verbo, url y cabecera para proporcionar información adicional (por ejemplo, el idioma de una página web). La respuesta contiene una línea inicial con el HTTP y un código asociado (200 si todo está bien, 404 si no se ha encontrado, etc), la cabecera y el cuerpo.



Los códigos de retorno son los siguientes:

Rango disponible	Rango definido	Categoría
100-199	100-101	Información
200-299	200-206	Éxito
300-399	300-305	Redirección
400-499	400-415	Error del cliente
500-599	500-505	Error del servidor

Tabla II.2: *Códigos de retorno*

Un servidor es un ordenador en un data center. Cada servicio tiene asignado un puerto, ya que en un mismo servidor puede haber varios servicios web. Las direcciones IP (técnicamente IPv4) se dividen en cuatro bloques de tres números que van del 0 al 255 (esto ocupa 8 bites). Así, en el mundo puede haber un máximo de 255^4 direcciones IP en el mundo. Por ello, a los servicios se les añade después de la dirección IP el puerto. En el caso de la web de la UAM, se trata de 150.244.214.237:80 para

HTTP y 150.244.214.237:443 para HTTP. Todos los servicios seguros (HTTPS) se conectan al puerto 443. Una vez conectados al servidor, se puede utilizar el método `GET` para obtener como respuesta la web que se está pidiendo. Esto se puede realizar en Python con la librería requests y la función get, como se explicó anteriormente. También está el método `POST` para enviar datos o información a un servidor. Estos datos se incluyen en el cuerpo de la petición, no están visibles en la URL, por lo que se suele utilizar para los login al ser más seguro (usuario y contraseña está oculto a la vista). En resumen: se pide información con GET y se envía información con POST.

Ejemplo: https://rest.ensembl.org/ Aquí se pueden ver todos los endpoints (las distintas funcionalidades) de la página Ensembl. Con `GET archive/id/:id` se puede acceder a la última versión del identificador que se dé. :id indica la variable que se debería proporcionar, es decir, sustituirlo con el ID. Lo que se devuelve está en formato JSON, pero al leerlo en Python será del tipo string. En este ejemplo, hay un endpoint `POST archive/id` que lo que hace es devolver la última versión para un conjunto de identificadores. Antes hemos dicho que POST se utiliza para enviar información y GET para obtenerla, pero esta API está mal diseñada y diferencia GET y POST en que el primero es para un solo identificador y el segundo para varios.

En esta página se proponen ejemplos de peticiones. Para GET sería la siguiente:

```
import requests, sys

server = "https://rest.ensembl.org"
ext = "/archive/id/ENSG00000157764?"

r = requests.get(server+ext, headers={ "Content-Type" :
    "application/json"})

if not r.ok:
  r.raise_for_status()
  sys.exit()

decoded = r.json()
print(repr(decoded))
```

II.3.3. APIs con HTTP

Los métodos HTTP originales tienen la siguiente semántica:

- GET: pide obtener una representación del recurso identificado
- POST: pide añadir información al recurso
- HEAD: como GET, pero sin recibir el cuerpo, solo la cabecera
- Otros verbos como DELETE, PUT, CONNECT, TRACE, etc.

Para nuestros fines, se utilizará casi siempre GET, pero algunos servidores de datos biomédicos también aceptan POST. Sin embargo, algunos servidores que aceptan

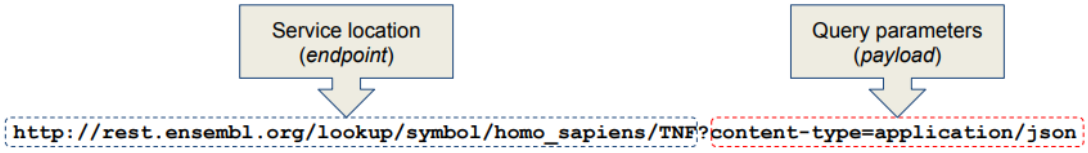
POST no lo utilizan para el fin previsto (crear información en el servidor), sino sólo para responder con la representación del recurso (al igual que GET). URLs como las siguientes utilizan el método GET:
https://rest.ensembl.org/sequence/id/ENSG00000157764
https://www.ebi.ac.uk/Tools/dbfetch/dbfetch?db=uniprot&id=WAP_RAT

Un método POST equivalente utiliza una URL sin parámetros (http://rest.ensembl.org/sequence/id/) que se deben enviar junto a los parámetros en el cuerpo de la petición ("ids":["ENSG00000157764"]).

```
## GET ##
import requests
gene = "ENSG00000157764"
endpoint = f"http://rest.ensembl.org/sequence/id/{gene}"
r = requests.get(endpoint, headers={ "Content-Type" :"text/plain"})

#Especificando params
gene = "ENSG00000157764"
endpoint = f"http://rest.ensembl.org/sequence/id/{gene}"
parameters = {'species': 'homo_sapiens'}
r = requests.get(endpoint, params = parameters, headers={"Content-Type"
    : "text/json"})

## POST ##
#Especificando data en lugar de params
response = requests.post(server, data = {'key':'value'})
```



Aunque las bases de datos sean de acceso público (open access), usualmente piden una API Key, que es un número largo único para cada usuario registrado que se pone en la cabecera. De esa forma, proporcionando el API Key, el servidor sabe qué usuario se está conectando y puede limitar las peticiones de un mismo usuario.

II.4. API REST

REST viene de REpresentational State Transfer. Realmente no es un protocolo, si no un estilo de prestación de servicios web basados en HTTP. Los **servidores** aportan representaciones textuales de recursos (como XML, JSON o HTML), y los **clientes** piden servicios mediante los métodos HTTP (GET, POST, etc). El estado no se guarda en el servidor (cada petición se responde de forma independiente sin memoria de peticiones anteriores).