Programación para la ciencia de datos

Unidad 1: Estructuras de datos avanzadas en Python

Instrucciones de uso

Este documento es un notebook interactivo que intercala explicaciones más bien teóricas de conceptos de programación con fragmentos de código ejecutables. Para aprovechar las ventajas que aporta este formato, se recomienda, en primer lugar, leer las explicaciones y el código que os proporcionamos. De esta manera tendréis un primer contacto con los conceptos que se exponen. Ahora bien, **¡la lectura es solo el principio!** Una vez que hayáis leído el contenido proporcionado, no olvidéis ejecutar el código proporcionado y modificarlo para crear variantes, que os permitan comprobar que habéis entendido su funcionalidad y explorar los detalles de la implementación. Por último, se recomienda también consultar la documentación enlazada para explorar con más profundidad las funcionalidades de los módulos presentados.

In [1]:

```
%load_ext pycodestyle_magic
```

In [2]:

```
# Activamos las alertas de estilo
%pycodestyle_on
```

Introducción

En esta unidad se presentan estructuras de datos y tipos de datos avanzados, y se hace una introducción a la manipulación avanzada de cadenas de caracteres.

En primer lugar, se explican estructuras de datos avanzadas que amplían las estructuras que ya conocíamos (listas, tuplas y diccionarios). Veremos qué son y cómo utilizar en Python los conjuntos, las pilas, las colas, los diccionarios con orden y los diccionarios con valores por defecto.

En segundo lugar, veremos tipos de datos complejos. En concreto, presentaremos tipo de datos que permiten representar fechas y horas, y explicaremos las funciones que Python implementa sobre estos tipos.

En tercer lugar y para terminar, nos centraremos en la manipulación avanzada de cadenas de caracteres, repasando algunas de las funciones básicas sobre cadenas y introduciendo el lenguaje y uso de expresiones regulares.

A continuación se incluye la tabla de contenidos, que podéis utilizar para navegar por el documento:

1.- Estructuras de datos avanzadas

Hasta ahora hemos visto algunas de las estructuras de datos más básicas que se utilizan para programar en Python: las listas, las tuplas y los diccionarios. En esta unidad, presentaremos estructuras de datos más complejas, que nos permitirán afrontar otro tipo de problemas, como son los conjuntos, las pilas, las colas, los diccionarios con orden, y los diccionarios con valores por defecto.

1.1.- Conjuntos

Del mismo modo que las listas, los conjuntos son una colección de elementos posiblemente **heterogéneos** (es decir, que pueden ser de tipos diferentes). Ahora bien, a diferencia de las listas, los elementos de un conjunto **no tienen orden** (ni por tanto posición dentro del conjunto). Además, un conjunto **no puede tener elementos repetidos**. En esencia, esta estructura de datos implementa el concepto matemático de conjunto.

Python dispone de la clase `set` (https://docs.python.org/3.8/library/stdtypes.html#set-types-set-frozenset) que implementa el conjunto. Podemos crear un conjunto utilizando el constructor de la clase o bien utilizando llaves `{}` :

In [3]:

```
# Creamos un conjunto utilizando dos sintaxis diferentes
a_set = {"Beale ciphers", "Quipu", "Zimmermann Telegram", "Chaocipher"}
the_same_set = set(
    ["Beale ciphers", "Quipu", "Zimmermann Telegram", "Chaocipher"])

# Confirmamos que los dos conjuntos creados son iguales
print("Boths sets are equal: {}".format(a_set == the_same_set))
```

```
Boths sets are equal: True
```

Hay que tener cuidado si usamos llaves para definir los conjuntos, ya que estas también se utilizan para definir diccionarios. Si los conjuntos y diccionarios tienen elementos, la sintaxis nos determinará el tipo. En cambio, usar llaves para definir una colección vacía resultará en un diccionario vacío:

In [4]:

```
# Definimos dos diccionarios vacíos utilizando las dos sintaxis
an_empty_dict = {}
another_empty_dict = dict()
print("Using {{}} results in a {} equivalent to dict(): {}".
      format(type(an_empty_dict), an_empty_dict == another_empty_dict))

# Definimos un conjunto vacío
an_empty_set = set()
print("Creating an empty {}".format(type(an_empty_set)))
```

```
Using {} results in a <class 'dict'> equivalent to dict(): True
Creating an empty <class 'set'>
```

Los conjuntos no tienen orden y, por lo tanto, dos conjuntos son iguales si sus elementos también lo son:

In [5]:

```
# Creamos otro conjunto con los mismos elementos, pero usamos otra
# ordenación de los elementos al crear el conjunto
another_same_set = {"Zimmermann Telegram", "Quipu",
                    "Chaocipher", "Beale ciphers"}
print("Boths sets are equal: {}".format(a_set == another_same_set))
```

```
Boths sets are equal: True
```

Como las listas, los conjuntos disponen de la operación `len`, que devuelve el número de elementos del conjunto (su cardinalidad). Las operaciones `add` (https://docs.python.org/3.8/library/stdtypes.html#frozenset.add) y `remove` (https://docs.python.org/3.8/library/stdtypes.html#frozenset.remove) permiten añadir y eliminar elementos de un conjunto:

In [6]:

```
# Mostramos la cardinalidad del conjunto
print("The set is: {}".format(a_set))
print("The size of the set is: {}".format(len(a_set)))

# Eliminamos un elemento del conjunto
a_set.remove("Quipu")
print("After removing 'Quipu', the set is: {}".format(a_set))

# Añadimos un elemento al conjunto
a_set.add("The Magic Words are Squeamish Ossifrage")
print("After adding an element, the set is: {}".format(a_set))
```

```
The set is: {'Zimmermann Telegram', 'Quipu', 'Beale ciphers', 'Chaoc
ipher'}
The size of the set is: 4
After removing 'Quipu', the set is: {'Zimmermann Telegram', 'Beale c
iphers', 'Chaocipher'}
After adding an element, the set is: {'Zimmermann Telegram', 'Chaoci
pher', 'The Magic Words are Squeamish Ossifrage', 'Beale ciphers'}
```

Los conjuntos no tienen elementos repetidos:

In [7]:

```
print("The size of the set is: {}".format(len(a_set)))

# Añadimos ahora un elemento ya existente en el conjunto
a_set.add("Beale ciphers")

# La cardinalidad del conjunto no varía, ya que el elemento ya
# pertenecía al conjunto
print("The size of the set after adding 'Beale ciphers' is: {}"
      .format(len(a_set)))
```

```
The size of the set is: 4
The size of the set after adding 'Beale ciphers' is: 4
```

Ya hemos visto como la clase `set` (https://docs.python.org/3.8/library/stdtypes.html#set) implementa la comparación de igualdad de conjuntos. Adicionalmente, también implementa las operaciones básicas de conjuntos (unión, intersección y diferencia) y la diferencia simétrica (que devuelve los elementos que están sólo en uno de los dos conjuntos).

A continuación vemos un ejemplo de estas operaciones sobre tres conjuntos, que contienen los nombres de los estudiantes de programación en Python, Java, y C, de una escuela (asumiremos que el nombre identifica de manera única los estudiantes).

In [8]:

```
# Creamos los tres conjuntos
students_python = {"Anna", "Helena", "Marta", "Pol"}
students_java = {"Anna", "Teresa", "Helena"}
students_c = {"Marc"}

# Calculamos los alumnos que estudian Python y también Java
print("Python and Java:\t\t{}".format(
    students_python.intersection(students_java)
))

# Calculamos los alumnos que estudian Python pero no java
print("Python but not Java:\t\t{}".format(
    students_python.difference(students_java)
))

# Calculamos los alumnos que estudian Python o Java, pero no
# ambos lenguajes
print("Python or Java, but not both:\t{}".format(
    students_python.symmetric_difference(students_java)
))

# Calculamos los estudiantes que tiene la escuela (independientemente
# del lenguaje que estudian)
print("Any of the languages:\t\t{}".format(
    students_python.union(students_java).union(students_c)
))
```

```
Python and Java:                {'Helena', 'Anna'}
Python but not Java:            {'Pol', 'Marta'}
Python or Java, but not both:   {'Pol', 'Marta', 'Teresa'}
Any of the languages:           {'Pol', 'Marta', 'Marc', 'Teresa',
'Helena', 'Anna'}
```

Notad que los resultados des las operaciones entre conjuntos que hemos visto en la celda anterior son también conjuntos, por lo que podemos encadenar operaciones (como muestra el último ejemplo, donde se obtienen todos los estudiantes de la escuela).

Equivalentemente, podemos utilizar los operandos `|`, `&`, `-` y `^`:

In [9]:

```
print("Python and Java:\t\t{}".format(
    students_python & students_java))
print("Python but not Java:\t\t{}".format(
    students_python - students_java))
print("Python or Java, but not both:\t{}".format(
    students_python ^ students_java))
print("Any of the languages:\t\t{}".format(
    students_python | students_java | students_c))
```

```
Python and Java:                {'Helena', 'Anna'}
Python but not Java:            {'Pol', 'Marta'}
Python or Java, but not both:   {'Pol', 'Marta', 'Teresa'}
Any of the languages:           {'Pol', 'Marta', 'Marc', 'Teresa',
'Helena', 'Anna'}
```

Los conjuntos son iterables, por lo que los podemos recorrer como hemos visto en la unidad anterior:

In [10]:

```
# Recorremos el conjunto students_python
for element in students_python:
    print(element)
```

```
Helena
Anna
Pol
Marta
```

Hasta ahora hemos presentado las propiedades de los conjuntos y las operaciones principales que podemos realizar sobre estos con la clase `set` (https://docs.python.org/3.8/library/stdtypes.html#set). Además, hemos visto como los conjuntos son la estructura de datos a utilizar cuando queremos mantener una colección de elementos sin duplicados y donde el orden no tenga importancia. Los conjuntos, asimismo, se pueden utilizar como herramienta para resolver, de manera eficiente, cierto tipo de problemas.

Por un lado, los conjuntos son una de las maneras más rápidas de eliminar duplicados de una lista:

In [11]:

```
# Definimos una lista con elementos duplicados
a_list = ["Anna", "Helena", "Marta", "Pol", "Anna", "Teresa", "Helena"]
print("A list with duplicates:\t\t\t{}".format(a_list))

# Creamos una nueva lista sin duplicados, haciendo un cast de la lista inicial
# a conjunto, y convirtiendo el resultado en lista de nuevo
a_list_without_duplicates = list(set(a_list))
print("The same list without duplicates:\t{}".format(
    a_list_without_duplicates))
```

```
A list with duplicates:                 ['Anna', 'Helena', 'Marta',
'Pol', 'Anna', 'Teresa', 'Helena']
The same list without duplicates:       ['Pol', 'Marta', 'Teresa',
'Helena', 'Anna']
```

Además de eficiente, la solución anterior es sencilla y elegante, siguiendo lo que nos recomienda el `PEP20` (https://www.python.org/dev/peps/pep-0020/).

Los conjuntos en Python también se encuentran optimizados para hacer tests de pertenencia, que nos permiten comprobar si un determinado elemento se encuentra presente en un conjunto:

In [12]:

```
a_student = "Anna"

print("Python students:\t{}".format(students_python))
print("Anna studies Python?:\t{}\n".format(a_student in students_python))

print("C students:\t\t{}".format(students_c))
print("Anna studies C?:\t{}".format(a_student in students_c))
```

```
Python students:        {'Helena', 'Anna', 'Pol', 'Marta'}
Anna studies Python?:   True

C students:             {'Marc'}
Anna studies C?:        False
```

1.2.- Pilas y colas

Las pilas y las colas son unas estructuras de datos también similares a las listas, donde los elementos tienen orden y pueden estar repetidos. La característica que define las pilas y las colas es que los elementos sólo se pueden insertar y eliminar desde uno de los extremos.

Una **pila** (en inglés, *stack*) sólo permite añadir y borrar elementos encima de la pila (en inglés, hablamos del *top*), es decir, por la parte superior. Las pilas se conocen también como estructuras FILO (del inglés, *first in last out*), ya que el primer elemento que se añade a una pila será el último que saldrá de ella. Una analogía en el mundo físico de una pila puede ser una pila de libros que se encuentra dentro de una caja: el primer libro que sacaremos de la caja será el de lo alto de la pila, que será el último que hayamos puesto en la caja.

Una **cola** (en inglés, *queue*) sólo permite añadir elementos al final y eliminar elementos del inicio. Las colas se conocen también como estructuras FIFO (del inglés, *first in first out*), ya que el primer elemento que se añade a una cola será el primero en salir. En nuestro día a día encontramos colas en varias circunstancias, por ejemplo, cuando esperamos nuestro turno en la carnicería o para comprar entradas en el cine.

A diferencia de los conjuntos, que en Python representamos con el tipo propio `set` (https://docs.python.org/3.8/library/stdtypes.html#set), para trabajar con pilas y colas con Python utilizaremos el tipo lista (`list` (https://docs.python.org/3.8/library/stdtypes.html#list)). Este tipo dispone de una serie de operaciones que permiten emular el comportamiento de pilas y colas. Python también dispone de una clase específica para gestionar colas, la clase `queue` (https://docs.python.org/3.8/library/queue.html), que se utiliza en implementaciones *multithread* que veremos más adelante en la asignatura.

Para implementar una pila, trabajaremos sobre una lista, utilizando únicamente los métodos `append` y `pop` para añadir y quitar elementos:

In [13]:

```
def print_stack(s, msg_bef=None, msg_after=None):
    # Definimos la función auxiliar print_stack, que nos permitirá
    # visualizar una pila
    if msg_bef:
        print(msg_bef)

    if len(s) == 0:
        max_len = 10
    else:
        max_len = max([len(e) for e in s]) + 4

    print("|" + " "*(max_len+2) + "|")
    for e in s[::-1]:
        print("| " + e + " "*(max_len-len(e)) + " |")
    print("|" + "_"*(max_len+2) + "|")

    print(msg_after+"\n" if msg_after else "\n")
```

In [14]:

```
# Iniciamos una lista vacía, que utilizaremos como pila
stack = []
print_stack(stack, "Initial stack:")

# Añadimos 3 elementos a la pila, y vamos mostrando el contenido de
# la pila en cada momento
stack.append("The Art of Computer Programming")
print_stack(stack, "Stack after adding 1 element:")

stack.append("The Mythical Man-Month")
print_stack(stack, "Stack after adding a 2nd element:")

stack.append("Refactoring")
print_stack(stack, "Stack after adding a 3rd element:")
```

```
Initial stack:
|            |
|____________|


Stack after adding 1 element:
|                                      |
| The Art of Computer Programming      |
|______________________________________|


Stack after adding a 2nd element:
|                                      |
| The Mythical Man-Month               |
| The Art of Computer Programming      |
|______________________________________|


Stack after adding a 3rd element:
|                                      |
| Refactoring                          |
| The Mythical Man-Month               |
| The Art of Computer Programming      |
|______________________________________|
```

Al añadir elementos, estos se van apilando de manera que el último elemento añadido queda en el *top* de la pila. En el ejemplo anterior, el último elemento añadido, *Refactoring*, queda en el *top* de la pila.

In [15]:

```
# Eliminamos un elemento de la pila y mostramos como queda la pila
e = stack.pop()
print_stack(stack, "Stack after deleting one element:",
            "The element removed was: {}".format(e))
```

```
Stack after deleting one element:
|                                        |
| The Mythical Man-Month                 |
| The Art of Computer Programming        |
|________________________________________|
The element removed was: Refactoring
```

Al ejecutar `pop`, se elimina el elemento del *top* de la pila, en este caso, *Refactoring*. El último elemento añadido es pues el primero en salir.

In [16]:

```
# Eliminamos dos elementos más de la pila y mostramos como queda la pila
# en cada momento
e = stack.pop()
print_stack(stack, "Stack after deleting another element:",
            "The element removed was: {}".format(e))

e = stack.pop()
print_stack(stack, "Stack after deleting the last element:",
            "The element removed was: {}".format(e))
```

```
Stack after deleting another element:
|                                        |
| The Art of Computer Programming        |
|________________________________________|
The element removed was: The Mythical Man-Month

Stack after deleting the last element:
|            |
|____________|
The element removed was: The Art of Computer Programming
```

Si seguimos eliminando elementos, veremos como el primer elemento añadido a la pila, *The Art of Computer Programming*, es el último en salir (el comportamiento FILO, *first in last out* que comentábamos).

Finalmente, ejecutar un `pop` sobre una pila vacía, generará una excepción:

In [17]:

```
try:
    stack.pop()
except IndexError as e:
    print(e)
```

```
pop from empty list
```

Para implementar una cola, trabajaremos también sobre una lista, utilizando únicamente los métodos `append` y `pop(0)` para añadir y quitar elementos:

In [18]:

```
def print_queue(s, msg_bef=None, msg_after=None):
    # Definimos la función auxiliar print_queue, que nos permitirá
    # visualizar una cola
    if msg_bef:
        print(msg_bef)

    if len(s) == 0:
        max_len = 10
    else:
        max_len = sum([len(e)+2 for e in s]) + 1

    print("_"*(max_len+2)+"\n")
    print("  " + "  ".join(s))
    print("_"*(max_len+2))

    print(msg_after+"\n" if msg_after else "\n")
```

In [19]:

```
# Iniciamos una lista vacía, que utilizaremos como cola
queue = []
print_queue(queue, "Initial queue:")

# Añadimos 3 elementos a la cola, y vamos mostrando el contenido de
# la cola en cada momento
queue.append("A")
print_queue(queue, "Queue after adding 1 element:")

queue.append("B")
print_queue(queue, "Queue after adding a 2nd element:")

queue.append("C")
print_queue(queue, "Queue after adding a 3rd element:")
```

```
Initial queue:
_____________


_____________


Queue after adding 1 element:
______

  A
______

Queue after adding a 2nd element:
_________

  A  B
_________

Queue after adding a 3rd element:
_____________

  A  B  C
_____________
```

In [20]:

```
# Eliminamos un elemento de la cola y mostramos como queda la cola
e = queue.pop(0)
print_queue(queue, "Queue after deleting one element:",
            "The element removed was: {}".format(e))
```

```
Queue after deleting one element:
_________

  B  C
_________
The element removed was: A
```

Al ejecutar `pop(0)` , se elimina el primer elemento de la cola, en este caso, A, que es el primer elemento que hemos añadido. Así pues, la cola es una estructura FIFO (*first in first out*).

In [21]:

```
# Eliminamos dos elementos más de la cola y mostramos como queda la cola
# en cada momento
e = queue.pop(0)
print_queue(queue, "Queue after deleting another element:",
            "The element removed was: {}".format(e))

e = queue.pop(0)
print_queue(queue, "Queue after deleting the last element:",
            "The element removed was: {}".format(e))
```

```
Queue after deleting another element:
______

  C

______
The element removed was: B

Queue after deleting the last element:
__________


__________
The element removed was: C
```

El último elemento añadido es pues también el último en salir (C).

De nuevo, si intentamos eliminar un elemento sobre una cola vacía, se genera una excepción:

In [22]:

```
try:
    queue.pop(0)
except IndexError as e:
    print(e)
```

```
pop from empty list
```

1.3.- Diccionarios ordenados

Aunque a partir de la versión 3.6 de Python los diccionarios preservan el orden de inserción de los elementos, la estructura de datos no contempla el orden de los elementos a la hora de hacer comparaciones. Así, dos diccionarios con el mismo contenido insertado en orden diferente son considerados iguales. Es decir, dos diccionarios son iguales si su contenido lo es, independientemente del orden en que se haya insertado este contenido:

In [23]:

```
# Creamos dos diccionarios con el mismo contenido en orden diferente
dict_1 = {"a": 0, "b": 1}
dict_2 = {"b": 1, "a": 0}

# Mostramos el contenido de los diccionarios en el orden en que items()
# devuelve los elementos
print("dict_1:")
for k, v in dict_1.items():
    print("{}: {}".format(k, v))

print("\ndict_2:")
for k, v in dict_2.items():
    print("{}: {}".format(k, v))

# Comprobamos la igualdad de los dos diccionarios
print("\ndict_1 and dict_2 are equal: {}".format(dict_1 == dict_2))
```

```
dict_1:
a: 0
b: 1

dict_2:
b: 1
a: 0

dict_1 and dict_2 are equal: True
```

Si por el contrario, queremos que el orden de los elementos de un diccionario determine su igualdad a otro diccionario, podemos utilizar un diccionario ordenado, implementado por la clase `OrderedDict` (https://docs.python.org/3.8/library/collections.html#collections.OrderedDict) en Python, en la librería `collections` (https://docs.python.org/3.8/library/collections.html).

In [24]:

```
from collections import OrderedDict

# Creamos dos diccionarios ordenados con el mismo contenido en orden diferente
dict_1 = OrderedDict([("a", 0), ("b", 1)])
dict_2 = OrderedDict([("b", 1), ("a", 0)])

# Comprobamos la igualdad de los dos diccionarios
print("\ndict_1 and dict_2 are equal: {}".format(dict_1 == dict_2))
```

```
dict_1 and dict_2 are equal: False
```

Más allá de la diferencia en la implementación de la igualdad, la clase `OrderedDict` (https://docs.python.org/3.8/library/collections.html#collections.OrderedDict) también nos ofrece un par de métodos relativos al orden de los elementos que no teníamos disponibles en los diccionarios: `reversed` y `move_to_end` (https://docs.python.org/3.8/library/collections.html#collections.OrderedDict.move_to_end).

In [25]:

```
# Mostramos el contenido de dict_2 en el orden original
print("\ndict_2:")
for k in dict_2:
    print("{}: {}".format(k, dict_2[k]))

# Mostramos el contenido de dict_2 en orden inverso
print("\ndict_2:")
for k in reversed(dict_2):
    print("{}: {}".format(k, dict_2[k]))
```

```
dict_2:
b: 1
a: 0

dict_2:
a: 0
b: 1
```

In [26]:

```
# Creamos un diccionario ordenado
dict_4 = OrderedDict([("a", 0), ("b", 1), ("c", 2), ("d", 3)])

# Movemos el elemento de clave 'b' al final del diccionario
dict_4.move_to_end('b')

# Mostramos el diccionario modificado, con el elemento 'b' al final
print(dict_4)
```

```
OrderedDict([('a', 0), ('c', 2), ('d', 3), ('b', 1)])
```

Como apunte final, hay que tener en cuenta que en general un diccionario estándar de Python es más eficiente que un diccionario ordenado. Así pues, utilizaremos siempre diccionarios estándar, a no ser que necesitemos las funcionalidades de un diccionario ordenado para la lógica de nuestra aplicación.

1.4.- Otras variantes de diccionario

Otra variante del diccionario que a menudo es de utilidad es el `defaultdict` (https://docs.python.org/3.8/library/collections.html#collections.defaultdict). Este tipo de diccionario comparte las propiedades del diccionario tradicional (`dict` (https://docs.python.org/3.8/library/stdtypes.html#mapping-types-dict)), pero permite establecer un valor predeterminado a las entradas del diccionario no inicializadas:

In [27]:

```
# Creamos un diccionario a_trad_dict e intentamos mostrar el valor
# de una clave inexistente
a_trad_dict = {}
try:
    print(a_trad_dict["non_existing_key"])
except KeyError as e:
    print(e)
```

```
'non_existing_key'
```

In [28]:

```
from collections import defaultdict

# Creamos un diccionario defaultdict con valor por defecto una lista
# e intentamos mostrar el valor de una clave inexistente
a_defaultdict = defaultdict(list)
print(a_defaultdict["non_existing_key"])


def get_0():
    # Creamos un diccionario defaultdict con valor per defecto una función
    # que retorna 0 e intentamos mostrar el valor de una clave inexistente
    return 0


another_defaultdict = defaultdict(get_0)
print(another_defaultdict["non_existing_key"])
```

```
[]
0
```

Los `defaultdict` (https://docs.python.org/3.8/library/collections.html#collections.defaultdict) pueden ser útiles para evitar tener que inicializar las entradas de un diccionario manualmente.

Por ejemplo, supongamos que tenemos un texto, y queremos obtener una lista de las palabras agrupadas según su letra inicial. Podríamos usar un diccionario para almacenar esta agrupación: la clave del diccionario almacenaría la letra inicial, y el valor de cada clave sería una lista de las palabras que empiezan por esa letra. Así, sólo habría que tener entradas para las letras que son iniciales de alguna de las palabras del texto, y no para todas las posibles letras iniciales.

Una implementación básica podría usar un diccionario normal, procesando cada palabra una a una y comprobando en cada caso si ya tenemos una entrada en el diccionario para la letra inicial de la palabra:

In [29]:

```
words = "Beautiful is better than ugly. Explicit is better than implicit. "
words += "Simple is better than complex."

# Creamos el diccionario
words_by_first_letter = dict()

# Iteramos por cada palabra del texto
for word in words.split(" "):
    # Recuperamos la primera letra
    first_letter = word[0]

    if first_letter in words_by_first_letter:
        # Si ya existe la entrada del diccionario, añadimos la palabra
        # a la lista
        words_by_first_letter[first_letter].append(word)
    else:
        # Si no existe aún ninguna palabra con la misma inicial, creamos
        # la entrada en el diccionario, creando la lista de palabras para
        # aquella
        # inicial y añadiendo la palabra
        words_by_first_letter[first_letter] = [word]

# Mostramos el resultado
print(words_by_first_letter)
```

```
{'B': ['Beautiful'], 'i': ['is', 'is', 'implicit.', 'is'], 'b': ['be
tter', 'better', 'better'], 't': ['than', 'than', 'than'], 'u': ['ug
ly.'], 'E': ['Explicit'], 'S': ['Simple'], 'c': ['complex.']}
```

Observemos ahora cómo podemos simplificar el código usando un `defaultdict` (https://docs.python.org/3.8/library/collections.html#collections.defaultdict):

In [30]:

```
# Creamos el diccionario defaultdict
words_by_first_letter = defaultdict(list)

# Iteramos por cada palabra del texto
for word in words.split(" "):
    # Recuperamos la primera letra
    first_letter = word[0]
    # Añadimos la palabra en la llista de la inicial correspondiente
    words_by_first_letter[first_letter].append(word)

print(words_by_first_letter)
```

```
defaultdict(<class 'list'>, {'B': ['Beautiful'], 'i': ['is', 'is',
'implicit.', 'is'], 'b': ['better', 'better', 'better'], 't': ['tha
n', 'than', 'than'], 'u': ['ugly.'], 'E': ['Explicit'], 'S': ['Simpl
e'], 'c': ['complex.']})
```

Tened en cuenta que, en este caso, no necesitamos inicializar manualmente la lista.

2.- Tipos de datos avanzados: el módulo `datetime`

Ya conocemos los tipos de datos básicos más habituales en Python: `int` (https://docs.python.org/3.8/library/functions.html#int) y `float` (https://docs.python.org/3.8/library/functions.html#float), que nos permiten representar valores numéricos; `bool` (https://docs.python.org/3.8/library/functions.html#bool), para representar valores booleanos; y `str` (https://docs.python.org/3.8/library/stdtypes.html#text-sequence-type-str), para representar cadenas de caracteres. También hemos utilizado tipos compuestos, tales como `list` (https://docs.python.org/3.8/library/stdtypes.html#list), `tuple` (https://docs.python.org/3.8/library/stdtypes.html#tupla), `dict` (https://docs.python.org/3.8/library/stdtypes.html#dict) y `set` (https://docs.python.org/3.8/library/stdtypes.html#set). En esta sección, presentaremos otras clases que permiten representar tipos más complejos, tales como fechas y tiempos.

El módulo `datetime` (https://docs.python.org/3.8/library/datetime.html) de Python ofrece herramientas para trabajar con fechas y horas. Así, este módulo nos provee de herramientas para afrontar las problemáticas más habituales de la gestión de fechas y horas, como son la representación del tiempo en cadenas de caracteres con formatos diferentes, cálculos relativos a calendarios, cálculo de incrementos de tiempo, conversiones entre unidades de tiempo, gestión de husos horarios, etc. Todo esto facilita enormemente el trabajo de tratar con conjuntos de datos que contienen fechas y horas.

A modo de ejemplo, intentad pensar como implementaríais, con las librerías de Python que hemos presentado hasta ahora, pequeños programas que permitieran calcular:

1. A partir de una fecha concreta, el día de la semana (lunes, martes, ..., domingo) que era, es o será ese día.
2. Si los valores "2019-11-15" y "Friday, November 15, 2019" corresponden o no a la misma fecha.
3. Qué día será dentro de 30 días.
4. Si el año 2200 será o no un año bisiesto.

Todo este tipo de problemas son fácilmente implementables con las funciones que proporciona el módulo `datetime` (https://docs.python.org/3.8/library/datetime.html). Las clases principales de este módulo son `date` (https://docs.python.org/3.8/library/datetime.html#date-objects), que permite trabajar con fechas; `time` (https://docs.python.org/3.8/library/datetime.html#time-objects) con tiempos; `datetime` (https://docs.python.org/3.8/library/datetime.html#datetime-objects) con fechas completas que incluyen también el tiempo; y `timedelta` (https://docs.python.org/3.8/library/datetime.html#timedelta-objects) que procesa incrementos de tiempo.

2.1.- Creación de objetos que representan fechas y horas

Podemos crear objetos que representan fechas, horas o fechas con horas especificando las diferentes componentes de cada estructura manualmente. El ejemplo siguiente muestra cómo crear una fecha, una hora y una fecha con hora utilizando esta estrategia:

In [31]:

```
import datetime

# Creamos un objeto date que representa una fecha concreta
a_date = datetime.date(year=2017, month=10, day=1)
# De manera más concisa, podríamos hacer:
# a_date = datetime.date(2017, 1, 11)
print(a_date)

# Creamos un objeto time que representa una hora
a_time = datetime.time(hour=17, minute=14, second=42)
# De manera más concisa, podríamos hacer:
# a_time = datetime.time(17, 14, 42)
print(a_time)

# Creamos un objeto datetime, que representa una fecha con tiempo
a_datetime = datetime.datetime(year=2017, month=10, day=1,
                               hour=17, minute=14, second=42)
# De manera más concisa, podríamos hacer:
# a_datetime = datetime.datetime(2017, 1, 11, 17, 14, 42)
print(a_datetime)
```

```
2017-10-01
17:14:42
2017-10-01 17:14:42
```

Ahora bien, es muy habitual encontrar conjuntos de datos en que las fechas y horas no se encuentran desglosadas en año, mes, día, hora, minuto y segundo, sino que éstas se representan como cadenas de caracteres, que pueden seguir convenciones diferentes a la hora de expresar las fechas y las horas. Así, tendremos que poder crear los objetos a partir de cadenas de caracteres, y necesitaremos algún lenguaje para poder definir el formato de estas cadenas.

Las clases `datetime` (https://docs.python.org/3.8/library/datetime.html), `date` (https://docs.python.org/3.8/library/datetime.html#date-objects) y `time` (https://docs.python.org/3.8/library/datetime.html#time-objects) disponen del método `strptime` (https://docs.python.org/3.8/library/datetime.html#datetime.datetime.strptime) que permite crear un objeto `datetime` (https://docs.python.org/3.8/library/datetime.html), `date` (https://docs.python.org/3.8/library/datetime.html#date-objects) y `time` (https://docs.python.org/3.8/library/datetime.html#time-objects) a partir de una cadena de caracteres con la fecha y/o hora y de otra cadena de caracteres que informa del formato. A continuación se expone cómo crear un objeto `datetime` (https://docs.python.org/3.8/library/datetime.html) a partir de una cadena de caracteres. Toda esta funcionalidad puede ser utilizada también de manera análoga para crear objetos `date` (https://docs.python.org/3.8/library/datetime.html#date-objects) y `time` (https://docs.python.org/3.8/library/datetime.html#time-objects).

Así pues, podemos crear un objeto `datetime` (https://docs.python.org/3.8/library/datetime.html) a partir de la cadena de caracteres "2017-10-01 17:14:42" de la manera siguiente:

In [32]:

```
str_date = "2017-10-01 17:14:42"
a_datetime_from_str = datetime.datetime.strptime(str_date, "%Y-%m-%d %H:%M:%S")
print(a_datetime_from_str)
a_datetime == a_datetime_from_str
```

```
2017-10-01 17:14:42
```

Out[32]:

```
True
```

El primer parámetro de la llamada, "2017-10-01 17:14:42", informa de la fecha a representar, mientras que el segundo parámetro, ""%Y-%m-%d %H:%M:%S", informa del formato que utiliza la primera cadena. Para representar el formato, se utilizan unos comodines, indicados con el símbolo `%`, y otros caracteres no especiales (como el guión, `-` o los dos puntos, `:`). Así, en este caso, informamos que la cadena que representa la fecha y el tiempo está formada por:

- el año representado usando cuatro dígitos (`%Y`),
- un guión que separa el año del mes (`-`),
- el mes representado utilizando dos dígitos (`%m`),
- un guión que separa el mes del día (`-`),
- el día representado usando dos dígitos (`%d`),
- un espacio separando la fecha de la hora (``),
- la hora representada en formato 24 horas y utilizando dos dígitos (`%H`),
- los dos puntos separando la hora de los minutos (`:`),
- los minutos representados con dos dígitos (`%M`),
- los dos puntos separando los minutos de los segundos (`:`),
- y finalmente, los segundos representados con dos dígitos (`%S`).

Es importante notar que la especificación del formato nos permite evitar ningún tipo de ambigüedad a la hora de interpretar la fecha. Así, por ejemplo, podemos saber que se quiere representar el día 1 de octubre y no el día 10 de enero. Podríamos cambiar esta interpretación, modificando la cadena que expresa el formato, de manera que el mes y el día intercambiaran el orden:

In [33]:

```
a_date_from_str = datetime.datetime.strptime(str_date, "%Y-%d-%m %H:%M:%S")
print(a_date_from_str)
```

```
2017-01-10 17:14:42
```

Tened en cuenta que, en el ejemplo anterior, la cadena que representa la fecha es exactamente la misma, pero cambiando la especificación del formato, hemos hecho que se interprete de manera diferente, indicando ahora el 10 de enero.

La cadena de caracteres que hemos utilizado siempre utiliza el formato completo para expresar cualquier unidad de tiempo, anteponiendo ceros si es necesario. Así, para representar el primer día del mes, utiliza dos dígitos, `01` . Existen comodines para representar también las diferentes unidades sin anteponer el cero, así como para representar años utilizando sólo dos dígitos, meses utilizando el nombre del mes en lugar del número, horas en formato AM/PM, y ¡muchas otras variantes! Podéis encontrar la especificación completa del formato en la documentación oficial del módulo `datetime` (https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes). A continuación, se exponen algunos ejemplos adicionales:

In [34]:

```
# Cambiando el formato de los separadores
a_date_from_str = datetime.datetime.strptime(
    "2017/10/01 17*14*42", "%Y/%m/%d %H*%M*%S")
print(a_date_from_str)
```

```
2017-10-01 17:14:42
```

In [35]:

```
# Expresando el año con dos dígitos
a_date_from_str = datetime.datetime.strptime(
    "17-10-01 17:14:42", "%y-%m-%d %H:%M:%S")
print(a_date_from_str)
a_date_from_str = datetime.datetime.strptime(
    "95-10-01 17:14:42", "%y-%m-%d %H:%M:%S")
print(a_date_from_str)
```

```
2017-10-01 17:14:42
1995-10-01 17:14:42
```

Notad cómo, en el caso anterior, Python interpreta el siglo en función de los últimos dos dígitos del año.

In [36]:

```
# Incluyendo el día de la semana y expresando el mes con el nombre
a_date_from_str = datetime.datetime.strptime(
    'Sunday, 01 October 2017 17:14:42', "%A, %d %B %Y %H:%M:%S")
print(a_date_from_str)
```

```
2017-10-01 17:14:42
```

In [37]:

```
# Con el día de la semana y el mes abreviados
a_date_from_str = datetime.datetime.strptime(
    'Sun, 01 Oct 2017 17:14:42', "%a, %d %b %Y %H:%M:%S")
print(a_date_from_str)
```

```
2017-10-01 17:14:42
```

En informática, también se puede utilizar el *unix time stamp* como manera de medir el tiempo. El *unix time stamp* representa el tiempo como un entero, que cuenta los segundos transcurridos desde el 1 de enero de 1970. Podemos crear objetos que representan una fecha a partir de un *unix timestamp* con el método `fromtimestamp` (https://docs.python.org/3.8/library/datetime.html#datetime.date.fromtimestamp):

In [38]:

```
a_date_from_ts = datetime.date.fromtimestamp(1573840426)
print(a_date_from_ts)
```

```
2019-11-15
```

A menudo también nos interesará utilizar la fecha o tiempo actual en nuestras aplicaciones. Podemos crear objetos que representen el momento actual a través de las funciones `today` (https://docs.python.org/3.8/library/datetime.html#datetime.datetime.today) y `now` (https://docs.python.org/3.8/library/datetime.html#datetime.datetime.now):

In [39]:

```
# Creamos un objeto con la fecha actual
now_date = datetime.date.today()
print(now_date)

# Creamos un objeto con la fecha y hora actual
now_datetime = datetime.datetime.now()
print(now_datetime)
```

```
2020-03-10
2020-03-10 18:40:55.299736
```

2.2.- Mostrando la fecha y el tiempo en diferentes formatos

Hasta ahora hemos visto como el método `strptime` (https://docs.python.org/3.8/library/datetime.html#datetime.datetime.strptime) permite crear objetos `datetime` (https://docs.python.org/3.8/library/datetime.html), `date` (https://docs.python.org/3.8/library/datetime.html#date-objects) y `time` (https://docs.python.org/3.8/library/datetime.html#time-objects) a partir de una cadena de caracteres. De manera análoga, el método `strftime` (https://docs.python.org/3.8/library/datetime.html#datetime.datetime.strftime) permite obtener una cadena de caracteres que representa una fecha, hora, o fecha con hora en un formato concreto, que se especifica utilizando la misma sintaxis que `strptime` (https://docs.python.org/3.8/library/datetime.html#datetime.datetime.strptime).

In [40]:

```
# Creamos un objeto con la fecha y hora actual
now_datetime = datetime.datetime.now()

# Mostramos la fecha y hora utilizando 4 dígitos para los años, y 2
# para el mes, día, hora, minuto y segundos
print(now_datetime.strftime("%Y-%m-%d %H:%M:%S"))

# Invertimos el orden del día y el mes
print(now_datetime.strftime("%Y-%d-%m %H:%M:%S"))

# Cambiamos el formato de los separadores
print(now_datetime.strftime("%Y/%m/%d %H*%M*%S"))

# Expresando el año con dos dígitos
print(now_datetime.strftime("%y-%m-%d %H:%M:%S"))

# Incluyendo el día de la semana y expresando el mes con el nombre
print(now_datetime.strftime("%A, %d %B %Y %H:%M:%S"))

# Con el día de la semana y el mes abreviados
print(now_datetime.strftime("%a, %d %b %Y %H:%M:%S"))
```

```
2020-03-10 18:40:55
2020-10-03 18:40:55
2020/03/10 18*40*55
20-03-10 18:40:55
Tuesday, 10 March 2020 18:40:55
Tue, 10 Mar 2020 18:40:55
```

Además de utilizar `strftime` (https://docs.python.org/3.8/library/datetime.html#datetime.datetime.strftime) para obtener cadenas de caracteres que representen fechas, también nos puede ser de utilidad la función `isoformat` (https://docs.python.org/3.8/library/datetime.html#datetime.date.isoformat), que ya devuelve la cadena en formato ISO 8601 (año - mes - día, utilizando 4 dígitos para el año y 2 para el día y mes), sin necesidad de especificarlo manualmente:

In [41]:

```
# Mostramos la fecha y hora en formato ISO 8601
now_datetime.isoformat()
```

Out[41]:

```
'2020-03-10T18:40:55.310353'
```

2.3.- Trabajando con fechas y tiempos

Ya hemos visto una de las funcionalidades de los objetos que representan fechas y tiempo, que es la de interpretar y generar cadenas de caracteres que representan fechas en diferentes formatos. A continuación, veremos otras operaciones habituales sobre estos objetos.

Podemos acceder a las diferentes componentes que conforman una fecha con hora consultando los atributos de los objetos `datetime` (https://docs.python.org/3.8/library/datetime.html), `date` (https://docs.python.org/3.8/library/datetime.html#date-objects) y `time` (https://docs.python.org/3.8/library/datetime.html#time-objects). Nótese que, en este caso, el tipo de datos de los valores a los que accedemos es un entero, en vez de una cadena de caracteres como obteníamos con `strftime` (https://docs.python.org/3.8/library/datetime.html#datetime.datetime.strftime):

In [42]:

```
# Accedemos a las diferentes componentes de una fecha con hora
y = now_datetime.year
mo = now_datetime.month
d = now_datetime.day
h = now_datetime.hour
mi = now_datetime.minute
s = now_datetime.second

print("Year is {} and is an {}".format(y, type(y)))
print("Month is {} and is an {}".format(mo, type(mo)))
print("Day is {} and is an {}".format(d, type(d)))
print("Hour is {} and is an {}".format(h, type(h)))
print("Minute is {} and is an {}".format(mi, type(mi)))
print("Second is {} and is an {}".format(s, type(s)))
```

```
Year is 2020 and is an <class 'int'>
Month is 3 and is an <class 'int'>
Day is 10 and is an <class 'int'>
Hour is 18 and is an <class 'int'>
Minute is 40 and is an <class 'int'>
Second is 55 and is an <class 'int'>
```

Más allá del acceso a las componentes que conforman la fecha, también podemos acceder a otros datos de esta, como a qué semana del año pertenece o bien a qué día de la semana corresponde:

In [43]:

```
# Obtenemos una tupla de 3 valores: (año ISO, núm. de semana ISO,
# día de la semana ISO)
iso_cal = now_datetime.isocalendar()
print("The week is: {}".format(iso_cal[1]))
print("The day is: {}".format(iso_cal[2]))
```

```
The week is: 11
The day is: 2
```

In [44]:

```
# También podemos obtener únicamente el día de la semana con weekday()
print("The day is: {}".format(now_datetime.weekday()))
```

```
The day is: 1
```

Las dos alternativas utilizan convenciones diferentes para referirse a los días de la semana: `weekday` (https://docs.python.org/3.8/library/datetime.html#datetime.datetime.weekday) considera que el el lunes es un 0 y el domingo es un 6, mientras que `isocalendar` (https://docs.python.org/3.8/library/datetime.html#datetime.datetime.isocalendar) representa los lunes con un 1 y los domingos con 7.

Para convertir el valor numérico devuelto por `weekday` (https://docs.python.org/3.8/library/datetime.html#datetime.datetime.weekday) a una cadena de caracteres con el nombre del día de la semana, podemos utilizar el módulo `calendar` (https://docs.python.org/3.8/library/calendar.html):

In [45]:

```
# Importamos el módulo calendar
import calendar

# Obtenemos el nombre del día
print('Day of Week: ', calendar.day_name[now_datetime.weekday()])

# Alternativamente, también hubiéramos podido utilizar strftime para obtener la
# mismo dato
print(now_datetime.strftime("Short day name: %a"))
print(now_datetime.strftime("Full day name: %A"))
```

```
Day of Week:  Tuesday
Short day name: Tue
Full day name: Tuesday
```

La gestión del idioma en que se muestran los nombres de los días de la semana o los meses se configura con el módulo `locale` (https://docs.python.org/3.8/library/locale.html). Así, por ejemplo, si queremos obtener el nombre del día de la semana en castalán, haríamos:

In [46]:

```
# Importamos el módulo locale
import locale

# Especificamos el idioma español
locale.setlocale(locale.LC_ALL, 'ca_ES.UTF-8')

# Mostramos el nombre del día
print('Day of Week:', calendar.day_name[now_datetime.weekday()])
```

```
Day of Week: dimarts
```

Otra de las características que implementan los objetos de representación de fechas y horas que trabajamos en esta unidad es la posibilidad de compararlos:

In [47]:

```
# Importamos la función sleep, que permite introducir esperas en la ejecución
# de código
from time import sleep

# Obtenemos la fecha y hora actuales
now_1_dt = datetime.datetime.now()
# Esperamos 3 segundos
sleep(3)
# Obtenemos la fecha y hora actuales (después de la espera de 3 segundos)
now_2_dt = datetime.datetime.now()

# Comparamos las dos fechas
print('now_1_dt == now_2_dt: {}'.format(now_1_dt == now_2_dt))
print('now_1_dt < now_2_dt: {}'.format(now_1_dt < now_2_dt))
print('now_1_dt.date() == now_2_dt.date(): {}'.format(
    now_1_dt.date() == now_2_dt.date()))
```

```
now_1_dt == now_2_dt: False
now_1_dt < now_2_dt: True
now_1_dt.date() == now_2_dt.date(): True
```

Los dos valores de fecha y hora no son iguales (ya que hay tres segundos de diferencia) pero, en cambio, el día sí es el mismo (a menos que ejecutamos el código justo en el momento en que cambiamos de día) .

2.4.- Incrementos de tiempo

Otra de las clases que incorpora el módulo `datetime` (https://docs.python.org/3.8/library/datetime.html) es `timedelta` (https://docs.python.org/3.8/library/datetime.html#timedelta-objects). Esta clase nos permite representar incrementos de tiempo, así como manipular fechas y horas utilizando estos incrementos.

Fijémonos, por ejemplo, en las dos fechas que hemos creado anteriormente con diferencia de tres segundos. Hemos visto que las podíamos comparar, para saber cuál es anterior o si son iguales. Ahora bien, ¿cómo podríamos saber el tiempo que ha transcurrido entre una y la otra? Podríamos calcularlo manualmente a partir de ir recuperando las diferentes componentes de la fecha con hora, pero la clase `timedelta` (https://docs.python.org/3.8/library/datetime.html#timedelta-objects) ya implementa esta funcionalidad:

In [48]:

```
# Calculamos el tiempo transcurrido y el mostramos
delta = now_2_dt - now_1_dt
print("Datetimes differ in: {} seconds".format(delta))
print("Delta is: {}".format(type(delta)))
```

```
Datetimes differ in: 0:00:03.003333 seconds
Delta is: <class 'datetime.timedelta'>
```

El resultado de restar dos objetos `datetime` (https://docs.python.org/3.8/library/datetime.html#datetime-objects) es un objeto `timedelta` (https://docs.python.org/3.8/library/datetime.html#timedelta-objects) que representa el tiempo transcurrido. A continuación, veremos algunos ejemplos más de cómo se usan objetos `timedelta` (https://docs.python.org/3.8/library/datetime.html#timedelta-objects) para representar incrementos de tiempo:

In [49]:

```
# Creamos tres datetimes
dt_1 = datetime.datetime(year=2017, month=10, day=1,
                         hour=17, minute=14, second=42)

dt_2 = datetime.datetime(year=2017, month=12, day=1,
                         hour=17, minute=14, second=42)

dt_3 = datetime.datetime(year=2017, month=10, day=15,
                         hour=20, minute=16, second=42)

# Creamos timedeltas con las diferencias
tm_1 = dt_2 - dt_1
tm_2 = dt_3 - dt_1
tm_3 = dt_3 - dt_2
print("Timedelta between dt_1 and dt_2: {}".format(tm_1))
print("Timedelta between dt_1 and dt_3: {}".format(tm_2))
print("Timedelta between dt_2 and dt_3: {}".format(tm_3))
```

```
Timedelta between dt_1 and dt_2: 61 days, 0:00:00
Timedelta between dt_1 and dt_3: 14 days, 3:02:00
Timedelta between dt_2 and dt_3: -47 days, 3:02:00
```

Fijaos como al mostrar por pantalla el valor de un `timedelta` (https://docs.python.org/3.8/library/datetime.html#timedelta-objects), obtenemos una cadena de caracteres anotada (se incluye el número de días y la palabra `days`, seguida de la hora informada con hora:minutos:segundos).

Si lo preferimos, también podemos obtener el número de segundos que representa el `timedelta` (https://docs.python.org/3.8/library/datetime.html#timedelta-objects) como valor numérico (en este caso , un `float` (https://docs.python.org/3.8/library/functions.html#float)):

In [50]:

```
# Mostramos el número de segundos del tm_1
ts = tm_1.total_seconds()
print("Seconds elapsed: {}".format(ts))
print("Result is: {}".format(type(ts)))
```

```
Seconds elapsed: 5270400.0
Result is: <class 'float'>
```

También podemos construir un objeto `timedelta` (https://docs.python.org/3.8/library/datetime.html#timedelta-objects) para obtener fechas pasadas o futuras a partir de una fecha de referencia:

In [51]:

```
# Creamos un timedelta de 3 semanas
d = datetime.timedelta(weeks=3)

# Calculamos que día será dentro de tres semanas
future_3_weeks = now_1_dt + d
print("The date in three weeks will be: {}".format(future_3_weeks))

# Calculamos que día era hace tres semanas
past_3_weeks = now_1_dt - d
print("The date three weeks ago was: {}".format(past_3_weeks))
```

```
The date in three weeks will be: 2020-03-31 18:40:55.398361
The date three weeks ago was: 2020-02-18 18:40:55.398361
```

Se pueden crear objetos `timedelta` (https://docs.python.org/3.8/library/datetime.html#timedelta-objects) especificando semanas, días, horas, minutos, segundos, milisegundos o microsegundos:

In [52]:

```
# Creamos varios timedeltas
one_day = datetime.timedelta(days=1)
eleven_seconds = datetime.timedelta(seconds=11)
five_hours = datetime.timedelta(hours=5)
eleven_and_so_seconds = datetime.timedelta(seconds=11.56845)
one_microsecond = datetime.timedelta(microseconds=1)

# Operamos con los timedeltas
print("One day in the future: {}".format(now_1_dt + one_day))
print("One day in the past: {}".format(now_1_dt - one_day))
print("Adding one microsecond: {}".format(now_1_dt + one_microsecond))
```

```
One day in the future: 2020-03-11 18:40:55.398361
One day in the past: 2020-03-09 18:40:55.398361
Adding one microsecond: 2020-03-10 18:40:55.398362
```

2.5.- El calendario

En los ejemplos que hemos visto en el apartado anterior, hemos utilizado el módulo `calendar` (https://docs.python.org/3.8/library/calendar.html#module-calendar) para obtener el nombre del día de la semana correspondiente a una fecha concreta. Las funcionalidades de este módulo, sin embargo, no se limitan al cálculo del día de la semana. El módulo `calendar` (https://docs.python.org/3.8/library/calendar.html#module-calendar), como su nombre indica, nos provee de funcionalidades relativas al calendario.

Un ejemplo de las funcionalidades más evidentes de un calendario es mostrar el propio calendario.

In [53]:

```
# Mostramos el calendario del año 2020
print(calendar.calendar(2020))
```

```
                                  2020

      de gener                de febrer                de març
dl dt dc dj dv ds dg    dl dt dc dj dv ds dg    dl dt dc dj dv d
s dg
       1  2  3  4  5                    1  2
1
 6  7  8  9 10 11 12     3  4  5  6  7  8  9     2  3  4  5  6
7  8
13 14 15 16 17 18 19    10 11 12 13 14 15 16     9 10 11 12 13 1
4 15
20 21 22 23 24 25 26    17 18 19 20 21 22 23    16 17 18 19 20 2
1 22
27 28 29 30 31          24 25 26 27 28 29       23 24 25 26 27 2
8 29
                                                30 31

      d'abril                 de maig                 de juny
dl dt dc dj dv ds dg    dl dt dc dj dv ds dg    dl dt dc dj dv d
s dg
       1  2  3  4  5                 1  2  3     1  2  3  4  5
6  7
 6  7  8  9 10 11 12     4  5  6  7  8  9 10     8  9 10 11 12 1
3 14
13 14 15 16 17 18 19    11 12 13 14 15 16 17    15 16 17 18 19 2
0 21
20 21 22 23 24 25 26    18 19 20 21 22 23 24    22 23 24 25 26 2
7 28
27 28 29 30             25 26 27 28 29 30 31    29 30

     de juliol               d'agost                 de setembre
dl dt dc dj dv ds dg    dl dt dc dj dv ds dg    dl dt dc dj dv d
s dg
       1  2  3  4  5                    1  2        1  2  3  4
5  6
 6  7  8  9 10 11 12     3  4  5  6  7  8  9     7  8  9 10 11 1
2 13
13 14 15 16 17 18 19    10 11 12 13 14 15 16    14 15 16 17 18 1
9 20
20 21 22 23 24 25 26    17 18 19 20 21 22 23    21 22 23 24 25 2
6 27
27 28 29 30 31          24 25 26 27 28 29 30    28 29 30
                        31

     d'octubre               de novembre             de desembre
dl dt dc dj dv ds dg    dl dt dc dj dv ds dg    dl dt dc dj dv d
s dg
          1  2  3  4                    1        1  2  3  4
5  6
 5  6  7  8  9 10 11     2  3  4  5  6  7  8     7  8  9 10 11 1
2 13
12 13 14 15 16 17 18     9 10 11 12 13 14 15    14 15 16 17 18 1
9 20
19 20 21 22 23 24 25    16 17 18 19 20 21 22    21 22 23 24 25 2
6 27
26 27 28 29 30 31       23 24 25 26 27 28 29    28 29 30 31
                        30
```

In [54]:

```
# Mostramos el calendario de octubre de 2020
print(calendar.month(2020, 10))
```

```
   d'octubre 2020
dl dt dc dj dv ds dg
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

Entre otras funcionalidades, el módulo `calendar` (https://docs.python.org/3.8/library/calendar.html#module-calendar) nos permite saber si un año es o no bisiesto con la función `isleap` (https://docs.python.org/3.8/library/calendar.html#calendar.isleap), o bien contar el número de días bisiestos que hay entre dos años:

In [55]:

```
# Calculamos si el año 2020 es bisiesto
calendar.isleap(2020)
```

Out[55]:

```
True
```

In [56]:

```
# Calculamos el número de días bisiestos entre los años
# 2000 y 2050
calendar.leapdays(2020, 2050)
```

Out[56]:

```
8
```

2.6.- Recapitulando

Al inicio de esta sección, nos planteábamos un conjunto de preguntas relacionadas con la gestión de las fechas y las horas que parecían, a priori, poco directas de resolver. Como punto final de la sección, responderemos a estas preguntas utilizando lo aprendido en esta unidad, y constataremos como de fácil puede ser contestarlas con las herramientas adecuadas.

En primer lugar, nos planteábamos cómo obtener el día de la semana (lunes, martes, ..., domingo) correspondiente a una fecha concreta.

A partir de un objeto `date` (https://docs.python.org/3.8/library/datetime.html#date-objects) (o `datetime` (https://docs.python.org/3.8/library/datetime.html#datetime-objects)) que representa una fecha, hemos presentado dos maneras diferentes de obtener el día de la semana:

In [57]:

```
# Creamos un objeto date que representa una fecha concreta
a_date = datetime.date(year=2022, month=12, day=25)

# Alternativa 1: Obtenemos el día con .weekday() y el nombre del día
# con el método day_name del módulo calendar
print('Day of Week:', calendar.day_name[a_date.weekday()])

# Alternativa 2: Usamos strftime para obtener el nombre del día de la semana
print(a_date.strftime("Day of Week: %A"))
```

```
Day of Week: diumenge
Day of Week: diumenge
```

En segundo lugar, queríamos descubrir si las cadenas de caracteres "2019-11-15" y "Friday, November 15, 2019" correspondían o no a la misma fecha.

Podemos crear dos objetos `date` (https://docs.python.org/3.8/library/datetime.html#date-objects) que representen las fechas expresadas para cada una de las cadenas de caracteres y, después, compararlos los para comprobar su igualdad:

In [58]:

```
# Creamos dos objetos date a partir de las cadenas de caracteres, especificando
# como idioma el inglés para interpretar correctamente la segunda cadena
date_1 = datetime.datetime.strptime(
    "2019-11-15", "%Y-%m-%d")
locale.setlocale(locale.LC_ALL, 'en_GB.UTF-8')
date_2 = datetime.datetime.strptime(
    "Friday, November 15, 2019", "%A, %B %d, %Y")

# Comparamos la igualdad de las fechas creadas
date_1 == date_2
```

Out[58]:

```
True
```

En tercer lugar, queríamos saber qué día será dentro de 30 días.

Podemos combinar la funcionalidad de obtención de la fecha actual con un `timedelta` (https://docs.python.org/3.8/library/datetime.html#timedelta-objects) que represente 30 días para saber qué día será dentro de 30 días:

In [59]:

```
# Creamos un objeto con la fecha actual
now_date = datetime.date.today()

# Creamos un datetime de 30 días
dt_30days = datetime.timedelta(days=30)

# Sumamos los 30 días a la fecha actual
print(now_date + dt_30days)
```

```
2020-04-09
```

Finalmente, nos preguntábamos si el año 2200 será o no un año bisiesto.

El módulo `calendar` (https://docs.python.org/3.8/library/calendar.html#module-calendar) nos permite comprobar si un año es bisiesto con el método `isleap` (https://docs.python.org/3.8/library/calendar.html#calendar.isleap):

In [60]:

```
# Comprobamos si el año 2200 es un año bisiesto
calendar.isleap(2200)
```

Out[60]:

```
False
```

3.- Cadenas de caracteres

Ya conocemos algunas de las funciones de manipulación de cadenas de caracteres básicas, que nos permiten concatenar cadenas entre ellas, formar cadenas a partir de otras cadenas y de variables numéricas, particionar cadenas (usando *slicing*) y crear cadenas de caracteres a partir de listas. Iniciaremos esta sección haciendo un repaso de estas funciones básicas:

In [61]:

```
# Creamos 4 cadenas de caracteres
str_1 = "Today is:"
str_2 = "Friday"
str_3 = "Coding day"
pep_20 = "Beautiful is better than ugly.\nExplicit is better than implicit."
```

In [62]:

```
# Creamos nuevas cadenas concatenando las cadenas anteriores
# con espacios que las separan
str_4 = str_1 + " " + str_2
print(str_4)
str_5 = str_1 + " " + str_3
print(str_5)
```

```
Today is: Friday
Today is: Coding day
```

In [63]:

```
# Creamos cadenas a partir de valores de variables numéricas
str_6 = str_1 + " my {}th day of programming".format(6)
print(str_6)
```

```
Today is: my 6th day of programming
```

In [64]:

```
# También podemos utilizar la funcionalidad de las cadenas-f
# para formatear texto de manera compacta
days = 6
f'{str_1} my {days}th day of programming'
```

Out[64]:

```
'Today is: my 6th day of programming'
```

In [65]:

```
# Mostramos los 9 primeros caracteres de la cadena pep_20
print(pep_20[:9])
# Mostramos los caracteres del 25 al 29 (este último no incluido)
print(pep_20[25:29])
# Mostramos el último carácter
print(pep_20[-1:])
```

```
Beautiful
ugly
.
```

In [66]:

```
# Creamos una cadena a partir de una lista de cadenas
a_list_of_str = ["A", "B", "C", "D"]
"".join(a_list_of_str)
```

Out[66]:

```
'ABCD'
```

In [67]:

```
# Creamos una cadena de caracteres con los números del 0 al 9
# separados por espacios
" ".join([str(x) for x in list(range(10))])
```

Out[67]:

```
'0 1 2 3 4 5 6 7 8 9'
```

In [68]:

```
# Creamos una cadena de caracteres con los números del 0 al 9
# separados por comas y espacios
", ".join([str(x) for x in list(range(10))])
```

Out[68]:

```
'0, 1, 2, 3, 4, 5, 6, 7, 8, 9'
```

3.1- Manipulación de cadenas de caracteres

A la hora de procesar una cadena de caracteres, a menudo será necesario particionarla, para manipular los fragmentos de manera individual. La función `split` (https://docs.python.org/3.8/library/stdtypes.html#str.split) permite particionar una cadena de caracteres, obteniendo como resultado una lista de subcadenas de la cadena original.

Llamada sin argumentos, la función `split` (https://docs.python.org/3.8/library/stdtypes.html#str.split) utiliza caracteres en blanco (espacios, tabuladores, saltos de línea, etc.) como separadores, y devuelve una lista de las subcadenas separadas, omitiendo estos separadores:

In [69]:

```
# Separamos la cadena pep_20
print(pep_20)
pep_20.split()
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
```

Out[69]:

```
['Beautiful',
 'is',
 'better',
 'than',
 'ugly.',
 'Explicit',
 'is',
 'better',
 'than',
 'implicit.']
```

Tened en cuenta que las palabras 'ugly.' y 'Explicit' quedan separadas como dos subcadenas diferentes, ya que el salto de línea entre una y la otra se interpreta como separador. En cambio, si queremos que se separen las palabras considerando únicamente los espacios en blanco, podemos indicarlo añadiendo el argumento `sep` a la llamada:

In [70]:

```
# Separamos la cadena pep_20 considerando sólo los espacios
# como separadores
pep_20.split(sep=" ")
```

Out[70]:

```
['Beautiful',
 'is',
 'better',
 'than',
 'ugly.\nExplicit',
 'is',
 'better',
 'than',
 'implicit.']
```

Podemos utilizar cualquier carácter (o cadena) como separador:

In [71]:

```
# Separamos la cadena por las letras "a"
pep_20.split("a")
```

Out[71]:

```
['Be', 'utiful is better th', 'n ugly.\nExplicit is better th', 'n i
mplicit.']
```

In [72]:

```
# Separamos la cadena utilizando la cadena "better" como separador
pep_20.split("better")
```

Out[72]:

```
['Beautiful is ', ' than ugly.\nExplicit is ', ' than implicit.']
```

Otra de las operaciones de procesamiento de cadenas de uso común es hacer un recuento del número de apariciones de una subcadena dentro de una cadena:

In [73]:

```
# Contamos cuántas veces aparece la letra "a" en la cadena pep_20
pep_20_a_num = pep_20.count("a")
print("PEP20 string has {} 'a'".format(pep_20_a_num))
# Contamos cuántas veces aparece la subcadena "better" en la cadena pep_20
pep_20_bet_num = pep_20.count("better")
print("PEP20 string has {} 'better'".format(pep_20_bet_num))
# Contamos cuantos espacios en blanco tiene la cadena pep_20
pep_20_ws_num = pep_20.count(" ")
print("PEP20 string has {} whitespaces".format(pep_20_ws_num))
```

```
PEP20 string has 3 'a'
PEP20 string has 2 'better'
PEP20 string has 8 whitespaces
```

Hemos visto como contar el número de apariciones de una subcadena. ¿Como lo haríamos, pero, si quisiéramos saber dónde están estas apariciones? La función `find` (https://docs.python.org/3.8/library/stdtypes.html#str.find) nos devuelve el índice donde se inicia una subcadena, o devuelve `-1` si tal subcadena no existe:

In [74]:

```
# Buscamos el índice de la primera aparición de 'beter'
str_to_search = "beter"
ind_better = pep_20.find(str_to_search)
print("'beter' starts at position {}".format(ind_better))

# Buscamos el índice de la primera aparición de 'better'
str_to_search = "better"
ind_better = pep_20.find(str_to_search)
print("'better' starts at position {}".format(ind_better))
```

```
'beter' starts at position -1
'better' starts at position 13
```

La cadena `better` aparece dos veces dentro de la cadena almacenada en la variable `pep_20` . La función `find` (https://docs.python.org/3.8/library/stdtypes.html#str.find) devuelve el índice de la primera aparición. Si queremos encontrar los índices del resto de apariciones, se deberá indicar un segundo parámetro a la llamada de `find` (https://docs.python.org/3.8/library/stdtypes.html#str.find), especificando la posición de inicio de la búsqueda:

In [75]:

```
# Buscamos el índice de la segunda aparición de 'better'
ind2_better = pep_20.find(str_to_search, ind_better+1)
print("Next 'better' starts at position {}".format(ind2_better))
```

```
Next 'better' starts at position 43
```

In [76]:

```
# Buscamos el índice de la tercera aparición de 'better'
ind3_better = pep_20.find(str_to_search, ind2_better+1)
print("Next 'better' starts at position {}".format(ind3_better))
```

```
Next 'better' starts at position -1
```

Como `better` sólo aparece dos veces, la tercera vez que hacemos la búsqueda la llamada a `find` (https://docs.python.org/3.8/library/stdtypes.html#str.find) devuelve `-1` , indicando que ya no hay más apariciones.

En algunas aplicaciones, no estaremos interesados en contar el número de apariciones de una subcadena ni en saber dónde es esta subcadena, sino simplemente en sustituirla por alguna otra cadena. En estos casos, podemos utilizar `replace` (https://docs.python.org/3.8/library/stdtypes.html#str.replace), que reemplaza todas las apariciones de una subcadena por otra cadena:

In [77]:

```
# Reemplazamos los espacios en blanco por guiones
print(pep_20.replace(" ", "_"), "\n")
# Reemplazamos 'better' por 'worse'
print(pep_20.replace("better", "worse"), "\n")
# Eliminamos los espacios en blanco reemplazándolos una cadena vacía
print(pep_20.replace(" ", ""))
```

```
Beautiful_is_better_than_ugly.
Explicit_is_better_than_implicit.

Beautiful is worse than ugly.
Explicit is worse than implicit.

Beautifulisbetterthanugly.
Explicitisbetterthanimplicit.
```

En el último ejemplo se eliminan todos los espacios en blanco. A veces, sin embargo, nos interesará eliminar los espacios en blanco que hay delante de una cadena o detrás de esta (o ambos conjuntos de espacios). Para estas situaciones, encontramos las funciones `lstrip` (https://docs.python.org/3.8/library/stdtypes.html#str.lstrip) (eliminar los espacios en blanco de la izquierda o el inicio de la cadena), `rstrip` (https://docs.python.org/3.8/library/stdtypes.html#str.rstrip) (eliminar los espacios de la derecha o el final de la cadena) y `strip` (https://docs.python.org/3.8/library/stdtypes.html#str.strip) (elimina los espacios en blanco de delante y detrás de la cadena):

In [78]:

```
# Creamos una cadena con espacios en blanco
str_with_spaces = "   A lot of   spaces "

# Mostramos el resultado de aplicar rstrip, lstrip y strip en la cadena
print("String: #{}#\n".format(str_with_spaces))
print("rstrip: #{}#".format(str_with_spaces.rstrip()))
print("lstrip: #{}#".format(str_with_spaces.lstrip()))
print("strip:  #{}#".format(str_with_spaces.strip()))
```

```
String: #   A lot of   spaces #

rstrip: #   A lot of   spaces#
lstrip: #A lot of   spaces #
strip:  #A lot of   spaces#
```

Es interesante notar que las funciones `strip` (https://docs.python.org/3.8/library/stdtypes.html#str.strip), `rstrip` (https://docs.python.org/3.8/library/stdtypes.html#str.rstrip) y `lstrip` (https://docs.python.org/3.8/library/stdtypes.html#str.lstrip) eliminan todos los espacios en blanco que se encuentran en la posición que están procesando (delante, detrás o en las dos posiciones) pero, en cambio, no tienen ningún efecto sobre el resto de espacios de la cadena (por ejemplo, el espacio en blanco entre 'A' y 'lot' nunca se ve afectado por la ejecución de estas funciones).

3.2- Expresiones regulares

A pesar de que las funciones que hemos visto hasta ahora son muy potentes, se quedan cortas para afrontar muchos de los problemas que encontraremos a la hora de preprocesar datos que contengan cadenas de texto.

Por ejemplo, si quisiéramos separar una cadena particionando por cualquier vocal (en vez de sólo la letra 'a' como hemos hecho en el ejemplo anterior), ¿como lo haríamos? Con lo que hemos visto hasta ahora, deberíamos ir separando vocal a vocal, procesando en cada caso todas las subcadenas ya separadas por las vocales anteriores.

Otro ejemplo sería si quisiéramos buscar no sólo la palabra 'better' dentro de una cadena, sino cualquier palabra de seis letras, o tal vez buscar palabras que empezaran con 'b' y acabaran con 'tter', con cualquier letra o letras entre la 'b' y 'tter'. ¿Como podríamos implementarlo con las funciones que hemos visto hasta ahora? Ciertamente, sería posible, pero el código sería complejo.

Para afrontar este tipo de problemas, se suelen utilizar **expresiones regulares** (también conocidas como *regex*, una contracción del término en inglés, *regular expressions*). Una expresión regular es una expresión que utiliza un lenguaje especializado de descripción de cadenas de caracteres. Este lenguaje nos permite expresar con facilidad condiciones que describen cómo debe ser una cadena como las que hemos ejemplificado anteriormente (una cadena que represente cualquier vocal, una cadena de seis letras, una cadena que empiece con 'b' y termine con 'tter', etc.). Estas expresiones se pueden usar, después, para tareas como las que hemos expuesto anteriormente: buscar subcadenas dentro de cadenas, separar cadenas, sustituir subcadenas por otros subcadenas, etc.

En este apartado, veremos pues como podemos utilizar expresiones regulares para procesar cadenas de caracteres. En primer lugar, veremos las funciones que nos ofrece el módulo `re` (https://docs.python.org/3.8/library/re.html). En segundo lugar, presentaremos una introducción a la sintaxis de las expresiones regulares.

3.2.1- Funciones del módulo `re`

El módulo `re` (https://docs.python.org/3.8/library/re.html) es la interfaz del motor de expresiones regulares para Python. Por lo tanto, habrá que importarlo cuando queramos trabajar con *regex*.

El módulo tiene funciones equivalentes a las que hemos visto anteriormente, pero que trabajan con expresiones regulares en vez de trabajar con cadenas de caracteres:

- `split` (https://docs.python.org/3.8/library/re.html#re.split): tiene el mismo comportamiento que el método `split` (https://docs.python.org/3.8/library/stdtypes.html#str.split) de `str` (https://docs.python.org/3.8/library/stdtypes.html#str), permitiendo separar cadenas considerando un separador, que en este caso estará expresado con una expresión regular.
- `search` (https://docs.python.org/3.8/library/re.html#re.search): tiene un comportamiento similar a `find` (https://docs.python.org/3.8/library/stdtypes.html#str.find) de `str` (https://docs.python.org/3.8/library/stdtypes.html#str), permitiendo encontrar la primera aparición de una subcadena dentro de una cadena. En este caso, la subcadena a buscar se expresa como una expresión regular.
- `sub` (https://docs.python.org/3.8/library/re.html#re.sub): tiene un comportamiento similar a `replace` (https://docs.python.org/3.8/library/stdtypes.html#str.replace) de `str` (https://docs.python.org/3.8/library/stdtypes.html#str), permitiendo reemplazar subcadenas por otra cadena. En este caso, la subcadena a reemplazar se expresa como una expresión regular.

Adicionalmente, también incorpora otras funciones que son de utilidad, tales como:

- `findall` (https://docs.python.org/3.8/library/re.html#re.findall): que permite buscar todas las apariciones (no solapadas) de una subcadena dentro de otra cadena.

A continuación, veremos cómo podemos utilizar estas funciones sin aprovechar la funcionalidad de las expresiones regulares (que presentaremos en el siguiente apartado).

La función `split` (https://docs.python.org/3.8/library/re.html#re.split) actúa de manera análoga a la `split` (https://docs.python.org/3.8/library/stdtypes.html#str.split) de `str` (https://docs.python.org/3.8/library/stdtypes.html#str), devolviendo una lista con las subcadenas separadas por un separador:

In [79]:

```
# Importamos el módulo re
import re

# Separamos la cadena por las letras "a"
re.split('a', pep_20)
```

Out[79]:

```
['Be', 'utiful is better th', 'n ugly.\nExplicit is better th', 'n i
mplicit.']
```

Ahora, sin embargo, la primera cadena (`a`) podrá ser también una expresión regular, que nos describa cómo debe ser el separador a utilizar.

La función `search` (https://docs.python.org/3.8/library/re.html#re.search) busca la primera aparición de una subcadena en otra cadena. A diferencia de `find` (https://docs.python.org/3.8/library/stdtypes.html#str.find), no devuelve directamente el índice de inicio de la coincidencia, sino un objeto `match` (https://docs.python.org/3.8/library/re.html#match-objects) que contiene información adicional sobre la coincidencia:

In [80]:

```
# Buscamos el índice de la primera aparición de 'better'
str_to_search = "better"
srch_better = re.search(str_to_search, pep_20)
print("The returned object is: {}".format(type(srch_better)))
print("that contains: {}".format(srch_better))
print("Substring starts at {} and ends at {}".format(
    srch_better.start(), srch_better.end()))
```

```
The returned object is: <class '_sre.SRE_Match'>
that contains: <_sre.SRE_Match object; span=(13, 19), match='bette
r'>
Substring starts at 13 and ends at 19
```

En cambio, la función `findall` (https://docs.python.org/3.8/library/re.html#re.findall) nos devolverá una lista con todas las coincidencias de una expresión regular:

In [81]:

```
# Buscamos todas las apariciones de 'better'
srch_better = re.findall(str_to_search, pep_20)
print("The returned object is: {}".format(type(srch_better)))
print("that contains: {}".format(srch_better))
```

```
The returned object is: <class 'list'>
that contains: ['better', 'better']
```

Finalmente, la función `sub` (https://docs.python.org/3.8/library/re.html#re.sub) reemplaza todas las apariciones de una subcadena por otra:

In [82]:

```
# Reemplazar los espacios en blanco para guiones bajos
print(re.sub(" ", "_", pep_20))
```

```
Beautiful_is_better_than_ugly.
Explicit_is_better_than_implicit.
```

Hasta ahora hemos visto como el módulo `re` (https://docs.python.org/3.8/library/re.html) nos permite hacer las mismas operaciones que habíamos visto sobre cadenas de caracteres. Vamos a ver ahora cómo podemos explotar el potencial de este módulo utilizando el lenguaje de las expresiones regulares.

Las funciones que hemos visto aceptan un primer parámetro que representa la subcadena a buscar, y que se utiliza como separador (en `split` (https://docs.python.org/3.8/library/re.html#re.split)), como objetivo de la búsqueda (`search` (https://docs.python.org/3.8/library/re.html#re.search) o `findall` (https://docs.python.org/3.8/library/re.html#re.findall)) o bien como cadena a reemplazar (`sub` (https://docs.python.org/3.8/library/re.html#re.sub)). Más allá de ser una cadena de caracteres, este primer parámetro puede usar el lenguaje de las expresiones regulares para especificar condiciones complejas sobre las cadenas a buscar.

3.2.2- Sintaxis de las expresiones regulares

En una expresión regular distinguimos dos tipos de caracteres:

- Caracteres **normales**, que representan el propio carácter. Por ejemplo, una 'a' es una expresión regular válida que representa una cadena de caracteres de un solo carácter que es la propia letra 'a'.
- Caracteres **especiales**, que sirven o bien para representar grupos de caracteres o bien para explicar cómo hay que interpretar las expresiones que los rodean.

Así, por ejemplo, algunos caracteres especiales se utilizan para representar **grupos de caracteres**:

- `.` : representa cualquier carácter (por defecto, cualquier carácter excepto la nueva línea).
- `\d` : representa cualquier dígito decimal.
- `\s` : representa cualquier espacio en blanco.
- `\w` : representa caracteres que pueden formar parte de una palabra (letras mayúsculas y minúsculas, números, el guión bajo).
- `[]` : permite indicar un conjunto de caracteres, ya sea individualmente (`[abc]` permitiría representar las cadenas `a` , `b` o `c`), o como rango (`[A-Z]` representa todas las letras en mayúscula).

Otros caracteres especiales dan información sobre **cómo interpretar** las expresiones:

- `|` : Implementa el operador OR, y permite crear una expresión regular que coincida o bien con una expresión regular o bien con otra.

También hay un conjunto de caracteres especiales que permiten especificar **repeticiones** de expresiones regulares. Así, los caracteres siguientes indican las repeticiones de la expresión que las precede:

- `*` : 0 o más apariciones.
- `+` : 1 o más apariciones.
- `?` : 0 o 1 aparición.
- `{m}` : exactamente *m* apariciones.
- `{m, n}` : entre *m* y *n* apariciones. Si uno de los dos valores se omite, entonces sólo se informa del mínimo o del máximo de apariciones.

3.2.3- Uso de expresiones regulares en la manipulación de cadenas

Ahora que ya conocemos las funciones de manipulación de cadenas del módulo `re` (https://docs.python.org/3.8/library/re.html) y la sintaxis básica del lenguaje de las expresiones regulares, vamos a ver cómo podemos explotar el potencial de ambas funcionalidades juntas a través de una serie de ejemplos. En primer lugar, veremos cómo podemos resolver algunos de los problemas que hemos comentado anteriormente. En segundo lugar, veremos ejemplos adicionales de uso de expresiones regulares.

Si queremos separar una cadena de caracteres utilizando cualquier vocal como separador, podemos utilizar una expresión regular que represente todas las vocales y utilizarla con la función `split` (https://docs.python.org/3.8/library/re.html#re.split). Los caracteres especiales `[]` nos permiten expresar un conjunto de caracteres individuales, por lo que podemos usarlos para expresar las vocales:

In [83]:

```
print(pep_20)

# Separamos la cadena por cualquier vocal
re.split(r'[aeiouAEIOU]', pep_20)
```

```
Beautiful is better than ugly.
Explicit is better than implicit.
```

Out[83]:

```
['B',
 '',
 '',
 't',
 'f',
 'l ',
 's b',
 'tt',
 'r th',
 'n ',
 'gly.\n',
 'xpl',
 'c',
 't ',
 's b',
 'tt',
 'r th',
 'n ',
 'mpl',
 'c',
 't.']
```

Fijaos como la lista contiene varias cadenas vacías, que resultan de separar vocales contiguas. Por otra parte, es interesante notar también que usamos el prefijo 'r' para indicar la cadena de caracteres que conforma la expresión regular. El prefijo 'r' indica que la cadena debe interpretarse como literal *en bruto* (en inglés, *raw string literal*).

Si queremos separar las palabras, podemos utilizar el carácter especial `\s`, para tener en cuenta cualquier espacio en blanco como separador (tanto el espacio `` ` `` como el salto de línea `\n` se utilizarían como separador para la cadena `pep_20`):

In [84]:

```
re.split(r'\s', pep_20)
```

Out[84]:

```
['Beautiful',
 'is',
 'better',
 'than',
 'ugly.',
 'Explicit',
 'is',
 'better',
 'than',
 'implicit.']
```

En cambio, si queremos separar las frases, podríamos utilizar como separador un punto seguido de un espacio en blanco. Fijaos que incluimos el espacio en blanco después del punto, ya que sino la última frase quedaría separada en dos por el punto final (la frase en sí y una cadena vacía), y el salto de línea quedaría incluido en la segunda frase :

In [85]:

```
# Separamos la cadena por frases: las frases se separan por un punto
# El resultado incluye una cadena vacía al final, ya que el último punto
# se utiliza como separador, así como el salto de línea
re.split(r'\.', pep_20)
```

Out[85]:

```
['Beautiful is better than ugly', '\nExplicit is better than implici
t', '']
```

In [86]:

```
# Separamos la cadena por frases: las frases se separan por un punto seguido
# de un espacio en blanco
re.split(r'\.\s', pep_20)
```

Out[86]:

```
['Beautiful is better than ugly', 'Explicit is better than implici
t.']
```

Es importante notar que para representar el punto, hemos utilizado la expresión `\.` en vez de `.`. El motivo es que, como hemos visto, `.` es un carácter especial que coincide con cualquier carácter. Si, por el contrario, queremos que la expresión regular sólo coincida cuando incluya un punto (el carácter '.'), entonces habrá **escapar** la secuencia, utilizando la barra invertida o contrabarra (`\`).

Otro ejemplo que nos planteábamos era como buscar palabras que empiecen con 'b' y terminen con 'tter', con cualquier letra o letras en medio. Vamos a ver cómo podríamos enfocar este ejemplo. En primer lugar, veremos cómo buscar palabras que empiecen con 'b' y terminen con 'tter', con una única letra en medio:

In [87]:

```
# Creamos una variante de la cadena pep_20, que nos permitirá ejemplificar
# las diferentes casuísticas
pep_20_bis = "Beautiful is better than ugly.\nExplicit is better than " \
    "implicit. Butter Much Webetter. bitter. bYYYtter or btter. python."

# Mostramos las cadenas de ejemplo
print(pep_20)
print("\n")
print(pep_20_bis)
```

```
Beautiful is better than ugly.
Explicit is better than implicit.


Beautiful is better than ugly.
Explicit is better than implicit. Butter Much Webetter. bitter. bYYY
tter or btter. python.
```

Una primera aproximación sería definir una expresión regular que comience con 'b', siga con cualquier carácter que pueda formar parte de una palabra (que podemos expresar con el carácter especial `\w`), y termine con 'tter':

In [88]:

```
print(r"b\wtter in pep_20: {}".format(re.findall(r"b\wtter", pep_20)))
print(r"b\wtter in pep_20_bis: {}".format(re.findall(r"b\wtter", pep_20_bis)))
```

```
b\wtter in pep_20: ['better', 'better']
b\wtter in pep_20_bis: ['better', 'better', 'better', 'bitter']
```

La expresión parece devolver los resultados esperados para la cadena `pep_20` pero, en cambio, para la cadena `pep_20_bis` nos devuelve un 'better' adicional y no es capaz de identificar 'Butter'. Los motivos de este comportamiento son, por un lado, que por defecto las expresiones regulares son *case sensitive*, por lo que 'B' y 'b' se consideran cadenas diferentes (y, por lo tanto, 'Butter' no es una coincidencia válida); y, por otra parte, que no estamos especificando delimitadores de las palabras, por lo que 'Webetter' es una coincidencia válida para 'better'.

Si queremos que ignoren si las letras son mayúsculas o minúsculas a la hora de evaluar una expresión regular, podemos utilizar el flag `IGNORECASE` en la llamada:

In [89]:

```
print(r"b\wtter in pep_20_bis: {}".format(
    re.findall(r"b\wtter", pep_20_bis, re.IGNORECASE)))
```

```
b\wtter in pep_20_bis: ['better', 'better', 'Butter', 'better', 'bit
ter']
```

De este modo, 'Butter' se identifica correctamente.

Por otra parte, si queremos buscar palabras completas (que no formen parte de otras palabras) podemos especificar el(los) carácter(es) que esperamos encontrar delante y detrás de la palabra:

In [90]:

```
# Delimitamos las palabras por espacios en blanco
print(r" b\wtter  in pep_20_bis: {}".format(
    re.findall(r" b\wtter ", pep_20_bis, re.IGNORECASE)))
# Delimitamos las palabras por espacios en blanco o bien puntos
print(r"[ \.]b\wtter[ \.] in pep_20_bis: {}".format(
    re.findall(r"[ \.]b\wtter[ \.]", pep_20_bis, re.IGNORECASE)))
```

```
 b\wtter  in pep_20_bis: [' better ', ' better ', ' Butter ']
[ \.]b\wtter[ \.] in pep_20_bis: [' better ', ' better ', ' Butter
', ' bitter.']
```

En segundo lugar, veremos ahora cómo buscar palabras que empiecen con 'b' y terminen con 'tter', pero aceptando más de una letra en medio. A partir de la expresión que hemos utilizado para una sola letra entre las dos subcadenas, podemos indicar que aceptamos cualquier número de repeticiones de la letra entre 'b' y 'tter' con el modificador `*`, o bien exigir que, como mínimo, haya una con `+`:

In [91]:

```
print("Using *: {}".format(
    re.findall(r"[ \.]b\w*tter[ \.]", pep_20_bis, re.IGNORECASE)))
print("Using +: {}".format(
    re.findall(r"[ \.]b\w+tter[ \.]", pep_20_bis, re.IGNORECASE)))
```

```
Using *: [' better ', ' better ', ' Butter ', ' bitter.', ' bYYYtter
', ' btter.']
Using +: [' better ', ' better ', ' Butter ', ' bitter.', ' bYYYtter
']
```

Por último, podríamos utilizar una estrategia similar para identificar todas las palabras de seis letras, delimitándolas por espacios o puntos, y utilizando el carácter especial `\w` repetido exactamente seis veces:

In [92]:

```
# Repitiendo el carácter \w 6 veces explícitamente:
print("6 letter words: {}".format(
    re.findall(r"[ \.]\w\w\w\w\w\w[ \.]", pep_20_bis)))
# Especificando el número de repeticiones con {}:
print("6 letter words: {}".format(
    re.findall(r"[ \.]\w{6}[ \.]", pep_20_bis)))
```

```
6 letter words: [' better ', ' better ', ' Butter ', ' bitter.', ' p
ython.']
6 letter words: [' better ', ' better ', ' Butter ', ' bitter.', ' p
ython.']
```

Esta sección ha presentado una pequeña introducción al uso de expresiones regulares con el módulo `re` (https://docs.python.org/3.8/library/re.html) de Python. Ahora bien, hay muchos otros caracteres especiales y construcciones que se pueden hacer con el lenguaje de las expresiones regulares, y que nos pueden ser útiles para procesar cadenas de caracteres. Para continuar el aprendizaje del uso de expresiones regulares en Python, os recomendamos leer el *Regular Expression HOWTO* (https://docs.python.org/3/howto/regex.html#regex-howto) de la documentación oficial de Python.

Adicionalmente, si queréis descubrir todos los detalles sobre el lenguaje de expresiones regulares y sobre las funciones que implementa el módulo `re` de Python, podéis dirigiros a la página oficial del módulo `re` (https://docs.python.org/3/library/re.html).

4.- Ejercicios para practicar

A continuación encontraréis un conjunto de problemas que pueden servir para practicar los conceptos explicados en esta primera unidad, así como para refrescar los conceptos básicos de programación. Os recomendamos que intentéis realizar estos problemas vosotros mismos y que, una vez realizados, comparéis la solución que proponemos con vuestra solución. No dudéis en dirigir todas las dudas que surjan de la resolución de estos ejercicios o bien de las soluciones propuestas al foro del aula.

1. Decidid cuál es la estructura de datos más adecuada para responder a cada una de las preguntas siguientes y escribid el código que permita responderlas.

Disponemos de datos de un conjunto de ciudades del mundo. De cada ciudad, sabemos si tiene más de 14 millones de habitantes, si es capital del país donde se encuentra, y si tiene una densidad de población por encima de los 20 000 habitantes por km^2:

- Las ciudades Shanghai, Beijing, Delhi, Estambul, Karachi, Guangzhou y Kinshasa tienen más de 14 millones de habitantes. El resto de ciudades de las que tenemos datos tienen 14 millones o menos de habitantes.
- Las ciudades Delhi, Beijing, Kinshasa, Tokio, Moscow, Jakarta, Seoul y Cairo son capitales del país donde se encuentran. El resto de ciudades de las que tenemos datos no son capitales.
- Las ciudades Cairo, Kinshasa, Delhi y Tokio tienen una densidad de población por encima de los 20 000 habitantes por km^2. El resto de ciudades de las que tenemos datos no superan los 20 000 habitantes por km^2.

1.1. ¿De cuántas ciudades (diferentes) tenemos datos? Asumiremos que no hay ninguna ciudad que no cumpla al menos una de las propiedades anteriores.

In [93]:

```
# Respuesta
```

1.2. ¿Cuántas ciudades tienen más de 14 millones de habitantes y una densidad de población por encima de los 20 000 habitantes por km^2?

In [94]:

```
# Respuesta
```

1.3. ¿Qué ciudades tienen una densidad de población por encima de los 20 000 habitantes por km^2 pero no más de 14 millones de habitantes?

In [95]:

```
# Respuesta
```

1.4. ¿Cuál es el país con mayor número de ciudades por encima de 14 millones de habitantes? ¿Cuántas ciudades de estas características hay en cada país?

Para responder a estas preguntas, nos faltará añadir información al conjunto de datos de ciudades disponible para realizar la actividad. Pensad cuál sería la estructura de datos más adecuada para almacenar esta información extra y calcular la respuesta a la pregunta planteada.

In [96]:

```
# Respuesta
```

1.5. ¿Cuáles son los dos países adyacentes que tienen el mayor número de ciudades por encima de 14 millones de habitantes?

Para responder a esta pregunta, nos faltará añadir más información al conjunto de datos de ciudades disponible para realizar la actividad. De nuevo, pensad cuál sería la estructura de datos más adecuada para almacenar esta información extra y calcular la respuesta a la pregunta planteada.

In [97]:

```
# Respuesta
```

1. Calculad cuántas horas ha trabajado la persona que ha escrito la siguiente frase:

"I started working at 17:22:42 and finished at 22:00:00"

In [98]:

```
sentence = "I started working at 17:22:42 and finished at 22:00:00"

# Respuesta
```

1. Dada la cadena de caracteres `sentence` , reemplazad todos los espacios en blanco por puntos.

In [99]:

```
sentence = " From     time to time, Python makes  an    incompatible change " \
    " to the   advertised semantics of core language constructs   "

# Respuesta
```

1. Dada la misma cadena de caracteres del ejercicio anterior, reemplazad todos los espacios en blanco contiguos por un único punto. Es decir, si encontráis tres espacios en blanco consecutivos, estos se deben reemplazar por un único punto, y no por tres puntos como implementábamos en el ejercicio anterior. Eliminad los espacios que se encuentran al inicio y al final de la cadena antes de sustituirlos por puntos.

In [100]:

```
# Respuesta
```

1. Proporcionad una lista con todas las palabras de cuatro letras de la cadena de caracteres `sentence` que empiecen por `t` o `F`.

In [101]:

```
# Respuesta
```

1. Reemplazad todas las mayúsculas de la cadena `sentence` por interrogantes.

In [102]:

```
# Respuesta
```

1. Una aerolínea nos contrata para ayudarla a optimizar el procedimiento de embarque de sus aviones. La aerolínea dispone de tres clases de billetes, primera, *business* y turista. A la hora de embarcar los clientes se sitúan en tres colas, una para cada clase. Después, sin embargo, sólo hay dos azafatas que les validen la tarjeta de embarque, por lo que las tres colas iniciales se convierten en dos colas, a partir de las cuales los pasajeros acceden a los aviones.

Actualmente, la aerolínea utiliza la siguiente estrategia para convertir las tres colas iniciales (por clase) en las dos colas que embarcan (una por cada azafata):

Por un lado, los clientes de primera y *business* van a parar a la cola 1 (cola prioritaria), intercalando un cliente de cada tipo en la cola prioritaria siempre que haya suficientes clientes para hacerlo, y asignando después todos los clientes restantes a la nueva cola. Así, el primer cliente que se asigna a la cola prioritaria es el primer cliente de la clase primera, el segundo cliente de la cola prioritaria será el primer cliente de la cola de *business*, el tercer cliente de la cola prioritaria será el segundo cliente de la clase primera, etc.

Por otra parte, los clientes de clase turista van a parar a la cola 2 (cola no prioritaria), siguiendo el orden que tenían en la cola de la clase turista. Ahora bien, si un cliente de clase turista tiene movilidad reducida o va acompañado de niños, entonces este se sitúa al frente de la cola prioritaria. Si hay más de un cliente en estas condiciones, el orden que siguen en la cola de la clase turista se mantiene.

Para evaluar cómo de buena es la estrategia, la aerolínea utiliza dos métricas:

1. El tiempo que se tarda en embarcar el avión es de 30 segundos por pasajero, considerando que las dos colas (1 y 2) embarcan a la vez. Es decir, si la cola 1 tiene un pasajero y la cola 2 tiene dos, se tardará un minuto en embarcar.
2. La satisfacción global de sus clientes, que se calcula haciendo la media de la satisfacción de cada cliente, considerando que:

- Los clientes de primera tienen una satisfacción de -25 veces el número de posiciones que han perdido en la cola prioritaria respecto a su posición original en la cola de primera. Es decir, un cliente que estaba en 3ª posición en la cola de primera y que ocupa la 5ª posición de la cola prioritaria, tendrá una satisfacción de -50.
- Los clientes de *business* siempre tienen una satisfacción de 0.
- Los clientes con billete de clase turista que han sido movidos a la cola prioritaria tienen una satisfacción de 100. En cambio, los que han sido movidos a la cola no prioritaria tienen una satisfacción de 25 si han avanzado alguna posición en la cola 2 respecto a su posición inicial en la cola de la clase turista, o de 0 en cualquier otro caso.

Como analistas de datos, evaluaremos la satisfacción y el tiempo de embarque del avión del vuelo 714.

7.1 Cargad los datos de los pasajeros del vuelo 714 que encontraréis en el dataset `data/flight714.csv` y cread las tres colas, primera, *business* y turista con los datos de los pasajeros. Los clientes se encuentran ordenados según su posición en la cola, con la columna `client_class` indicando a cuál de las colas pertenecen.

In [103]:

```
# Respuesta
```

7.2 Mostrad cuántos pasajeros hay en cada cola, con el detalle de cuántos de ellos tienen o bien criaturas o movilidad reducida.

In [104]:

```
# Respuesta
```

7.3 Implementad una función que genere las dos colas de embarque (cola prioritaria y cola no prioritaria) a partir de las tres colas obtenidas según la clase del billete del pasajero.

In [105]:

```
# Respuesta
```

7.4 Implementad una función que calcule el tiempo que se tarda en embarcar el avión y una función que calcule la satisfacción de los pasajeros.

In [106]:

```
# Respuesta
```

7.5 Calculad el tiempo que se tarda en embarcar el vuelo 714 y la satisfacción de los clientes.

In [107]:

```
# Respuesta
```

4.1.- Soluciones a los ejercicios para practicar

1. Decidid cuál es la estructura de datos más adecuada para responder a cada una de las preguntas siguientes y escribid el código que permita responderlas.

Disponemos de datos de un conjunto de ciudades del mundo. De cada ciudad, sabemos si tiene más de 14 millones de habitantes, si es capital del país donde se encuentra, y si tiene una densidad de población por encima de los 20 000 habitantes por km^2:

- Las ciudades Shanghai, Beijing, Delhi, Estambul, Karachi, Guangzhou y Kinshasa tienen más de 14 millones de habitantes. El resto de ciudades de las que tenemos datos tienen 14 millones o menos de habitantes.
- Las ciudades Delhi, Beijing, Kinshasa, Tokio, Moscow, Jakarta, Seoul y Cairo son capitales del país donde se encuentran. El resto de ciudades de las que tenemos datos no son capitales.
- Las ciudades Cairo, Kinshasa, Delhi y Tokio tienen una densidad de población por encima de los 20 000 habitantes por km^2. El resto de ciudades de las que tenemos datos no superan los 20 000 habitantes por km^2.

1.1. ¿De cuántas ciudades (diferentes) tenemos datos? Asumiremos que no hay ninguna ciudad que no cumpla al menos una de las propiedades anteriores.

In [108]:

```
# Respuesta

# Podemos guardar los datos en 3 conjuntos, uno para cada una de las
# propiedades que queremos analizar y calcular la unión de estos conjuntos

more_than_14M = {"Shanghai", "Beijing", "Delhi", "Istanbul", "Karachi",
                 "Guangzhou", "Kinshasa"}
capital = {"Delhi", "Beijing", "Kinshasa", "Tokyo", "Moscow", "Jakarta",
           "Seoul", "Cairo"}
more_than_20K = {"Cairo", "Kinshasa", "Delhi", "Tokyo"}

print("We have data from {} different cities".format(
    len(more_than_14M.union(capital).union(more_than_20K))))
```

```
We have data from 12 different cities
```

1.2. ¿Cuántas ciudades tienen más de 14 millones de habitantes y una densidad de población por encima de los 20 000 habitantes por km^2?

In [109]:

```
# Respuesta

# Podemos aprovechar los 2 conjuntos de interés (more_than_14M y more_than_20K)
# creados en el apartado anterior, y calcular la intersección.

print("There are {} cities with more than 14M inhabitants and a density "
      "over 20K/km^2".format(len(more_than_14M.intersection(more_than_20K))))
```

```
There are 2 cities with more than 14M inhabitants and a density over
20K/km^2
```

1.3. ¿Qué ciudades tienen una densidad de población por encima de los 20 000 habitantes por km^2 pero no más de 14 millones de habitantes?

In [110]:

```
# Respuesta

# De nuevo, podemos aprovechar los 2 conjuntos de interés (more_than_14M y
# more_than_20K) creados en el primer apartado, y calcular ahora la diferencia.

print("Cities with a density over 20K/km^2 and less than 14M inhabitants "
      "different cities: {}".format(more_than_20K.difference(more_than_14M)))
```

```
Cities with a density over 20K/km^2 and less than 14M inhabitants di
fferent cities: {'Tokyo', 'Cairo'}
```

1.4. ¿Cuál es el país con mayor número de ciudades por encima de 14 millones de habitantes? ¿Cuántas ciudades de estas características hay en cada país?

Para responder a estas preguntas, nos faltará añadir información al conjunto de datos de ciudades disponible para realizar la actividad. Pensad cuál sería la estructura de datos más adecuada para almacenar esta información extra y calcular la respuesta a la pregunta planteada.

Resposta:

In [111]:

```
# Respuesta

# Guardamos el país al que pertenece cada una de las ciudades con más
# de 14M de habitantes
city_country = {'Shanghai': 'China', 'Beijing': 'China',
                'Delhi': 'India', 'Istanbul': 'Turkey',
                'Karachi': 'Pakistan', 'Guangzhou': 'China',
                'Kinshasa': 'Congo'}

# Contamos cuántas ciudades de más de 14M de habitantes hay en cada país
cities_per_country = defaultdict(get_0)
for k, v in city_country.items():
    cities_per_country[v] += 1

print(cities_per_country)

# Recuperamos el país con más ciudades
# Opción 1: recorriendo el diccionario con un bucle
max_val = -1
city = None
for k, v in cities_per_country.items():
    if v > max_val:
        max_val = v
        city = k
print(city)
```

```
defaultdict(<function get_0 at 0x7faacc344bf8>, {'China': 3, 'Indi
a': 1, 'Turkey': 1, 'Pakistan': 1, 'Congo': 1})
China
```

In [112]:

```
# Opción 2: recuperamos la clave del diccionario con valor máximo
import operator
print(max(cities_per_country.items(), key=operator.itemgetter(1))[0])
```

```
China
```

1.5. ¿Cuáles son los dos países adyacentes que tienen el mayor número de ciudades por encima de 14 millones de habitantes?

Para responder a esta pregunta, nos faltará añadir más información al conjunto de datos de ciudades disponible para realizar la actividad. De nuevo, pensad cuál sería la estructura de datos más adecuada para almacenar esta información extra y calcular la respuesta a la pregunta planteada.

In [113]:

```
# Respuesta

cities_per_country.keys()
```

Out[113]:

```
dict_keys(['China', 'India', 'Turkey', 'Pakistan', 'Congo'])
```

In [114]:

```
# Guardamos las fronteras de cada país
# (omitimos los países que no están en el dataset)
country_frontier = {
    'China': ['India', 'Pakistan', 'Russia'],
    'India': ['China', 'Pakistan', 'Indonesia'],
    'Turkey': [],
    'Pakistan': ['China', 'India'],
    'Congo': []}

# Calculamos cuáles son los dos países adyacentes con más ciudades
# de más de 14M
cities_per_adj_countries = {}
countries = list(cities_per_country.keys())
for i in range(len(countries)):
    for j in range(i+1, len(countries)):
        country_1, country_2 = countries[i], countries[j]
        if country_1 in country_frontier[country_2]:
            cities_per_adj_countries[country_1 + '-' + country_2] = \
                cities_per_country[country_1] + cities_per_country[country_2]

cities_per_adj_countries
```

Out[114]:

```
{'China-India': 4, 'China-Pakistan': 4, 'India-Pakistan': 2}
```

1. Calculad cuántas horas ha trabajado la persona que ha escrito la siguiente frase:

"I started working at 17:22:42 and finished at 22:00:00"

In [115]:

```
sentence = "I started working at 17:22:42 and finished at 22:00:00"

# Respuesta

start = datetime.datetime.strptime(sentence[21:29], "%H:%M:%S")
end = datetime.datetime.strptime(sentence[-8:], "%H:%M:%S")
working_time = end - start

print("The person has worked {} hours".format(working_time))
```

```
The person has worked 4:37:18 hours
```

1. Dada la cadena de caracteres `sentence`, reemplazad todos los espacios en blanco por puntos.

In [116]:

```
sentence = " From     time to time, Python makes  an    incompatible change " \
    " to the   advertised semantics of core language constructs   "

# Respuesta

sentence.replace(" ", ".")
```

Out[116]:

```
'.From.....time.to.time,.Python.makes..an....incompatible.change..t
o.the...advertised.semantics.of.core.language.constructs...'
```

1. Dada la misma cadena de caracteres del ejercicio anterior, reemplazad todos los espacios en blanco contiguos por un único punto. Es decir, si encontráis tres espacios en blanco consecutivos, estos se deben reemplazar por un único punto, y no por tres puntos como implementábamos en el ejercicio anterior. Eliminad los espacios que se encuentran al inicio y al final de la cadena antes de sustituirlos por puntos.

In [117]:

```
# Respuesta

# Eliminamos los espacios en blanco
stripped_sentence = sentence.strip()
# Sustituimos los espacios en blanco por puntos
re.sub(" +", ".", stripped_sentence)
```

Out[117]:

```
'From.time.to.time,.Python.makes.an.incompatible.change.to.the.adver
tised.semantics.of.core.language.constructs'
```

1. Proporcionad una lista con todas las palabras de cuatro letras de la cadena de caracteres `sentence` que empiecen por `t` o `F`.

In [118]:

```
# Respuesta

re.findall(" [t|F]\w{3} ", sentence)
```

Out[118]:

```
[' From ', ' time ']

3:19: W605 invalid escape sequence '\w'
```

1. Reemplazad todas las mayúsculas de la cadena `sentence` por interrogantes.

In [119]:

```
# Respuesta

re.sub("[A-Z]", "?", sentence)
```

Out[119]:

```
' ?rom     time to time, ?ython makes  an    incompatible change  to
the   advertised semantics of core language constructs   '
```

1. Una aerolínea nos contrata para ayudarla a optimizar el procedimiento de embarque de sus aviones. La aerolínea dispone de tres clases de billetes, primera, *business* y turista. A la hora de embarcar los clientes se sitúan en tres colas, una para cada clase. Después, sin embargo, sólo hay dos azafatas que les validen la tarjeta de embarque, por lo que las tres colas iniciales se convierten en dos colas, a partir de las cuales los pasajeros acceden a los aviones.

Actualmente, la aerolínea utiliza la siguiente estrategia para convertir las tres colas iniciales (por clase) en las dos colas que embarcan (una por cada azafata):

Por un lado, los clientes de primera y *business* van a parar a la cola 1 (cola prioritaria), intercalando un cliente de cada tipo en la cola prioritaria siempre que haya suficientes clientes para hacerlo, y asignando después todos los clientes restantes a la nueva cola. Así, el primer cliente que se asigna a la cola prioritaria es el primer cliente de la clase primera, el segundo cliente de la cola prioritaria será el primer cliente de la cola de *business*, el tercer cliente de la cola prioritaria será el segundo cliente de la clase primera, etc.

Por otra parte, los clientes de clase turista van a parar a la cola 2 (cola no prioritaria), siguiendo el orden que tenían en la cola de la clase turista. Ahora bien, si un cliente de clase turista tiene movilidad reducida o va acompañado de niños, entonces este se sitúa al frente de la cola prioritaria. Si hay más de un cliente en estas condiciones, el orden que siguen en la cola de la clase turista se mantiene.

Para evaluar cómo de buena es la estrategia, la aerolínea utiliza dos métricas:

1. El tiempo que se tarda en embarcar el avión es de 30 segundos por pasajero, considerando que las dos colas (1 y 2) embarcan a la vez. Es decir, si la cola 1 tiene un pasajero y la cola 2 tiene dos, se tardará un minuto en embarcar.
2. La satisfacción global de sus clientes, que se calcula haciendo la media de la satisfacción de cada cliente, considerando que:

- Los clientes de primera tienen una satisfacción de -25 veces el número de posiciones que han perdido en la cola prioritaria respecto a su posición original en la cola de primera. Es decir, un cliente que estaba en 3ª posición en la cola de primera y que ocupa la 5ª posición de la cola prioritaria, tendrá una satisfacción de -50.
- Los clientes de *business* siempre tienen una satisfacción de 0.
- Los clientes con billete de clase turista que han sido movidos a la cola prioritaria tienen una satisfacción de 100. En cambio, los que han sido movidos a la cola no prioritaria tienen una satisfacción de 25 si han avanzado alguna posición en la cola 2 respecto a su posición inicial en la cola de la clase turista, o de 0 en cualquier otro caso.

Como analistas de datos, evaluaremos la satisfacción y el tiempo de embarque del avión del vuelo 714.

7.1 Cargad los datos de los pasajeros del vuelo 714 que encontraréis en el dataset `data/flight714.csv` y cread las tres colas, primera, *business* y turista con los datos de los pasajeros. Los clientes se encuentran ordenados según su posición en la cola, con la columna `client_class` indicando a cuál de las colas pertenecen.

In [120]:

```
# Respuesta

import pandas as pd

df = pd.read_csv("data/flight714.csv")

bq, pq, tq = [], [], []
col_names = list(df.columns)

for i in range(len(df)):
    c = {c: df.iloc[i][c] for c in col_names}
    if c['client_class'] == "p":
        pq.append(c)
    if c['client_class'] == "b":
        bq.append(c)
    if c['client_class'] == "t":
        tq.append(c)
```

7.2 Mostrad cuántos pasajeros hay en cada cola, con el detalle de cuántos de ellos tienen o bien criaturas o movilidad reducida.

In [121]:

```
# Respuesta

print("Passengers in first class: {}".format(len(pq)))
print("Passengers in business class: {}".format(len(bq)))
print("Passengers in tourist class: {}".format(len(tq)))

f_with_crm = sum([1 for p in pq
                  if p['has_children'] or p['has_reduced_mobility']])
b_with_crm = sum([1 for p in bq
                  if p['has_children'] or p['has_reduced_mobility']])
t_with_crm = sum([1 for p in tq
                  if p['has_children'] or p['has_reduced_mobility']])

print("Passengers in first class with children or reduced mob.: {}".
      format(f_with_crm))
print("Passengers in business class with children or reduced mob.: {}".
      format(b_with_crm))
print("Passengers in tourist class with children or reduced mob.: {}".
      format(t_with_crm))
```

```
Passengers in first class: 3
Passengers in business class: 10
Passengers in tourist class: 37
Passengers in first class with children or reduced mob.: 0
Passengers in business class with children or reduced mob.: 0
Passengers in tourist class with children or reduced mob.: 6
```

7.3 Implementad una función que genere las dos colas de embarque (cola prioritaria y cola no prioritaria) a partir de las tres colas obtenidas según la clase del billete del pasajero.

In [122]:

```
# Respuesta

from copy import copy


def compute_new_queues_alg_1(in_bq, in_pq, in_tq):
    # Copiamos los datos de las colas de entrada para no modificar los
    # valores de las variables de entrada
    bq, pq, tq = copy(in_bq), copy(in_pq), copy(in_tq)
    # Inicializamos las dos colas de salida vacías
    q1, q2 = [], []

    # Movemos los pasajeros de la clase turista
    while len(tq) != 0:
        client = tq.pop(0)
        if client["has_children"] or client["has_reduced_mobility"]:
            q1.append(client)
        else:
            q2.append(client)

    # Movemos al resto de pasajeros, alternando entre primera y business
    while len(bq) != 0 or len(pq) != 0:
        if len(pq) != 0:
            client = pq.pop(0)
            q1.append(client)
        if len(bq) != 0:
            client = bq.pop(0)
            q1.append(client)

    return q1, q2
```

7.4 Implementad una función que calcule el tiempo que se tarda en embarcar el avión y una función que calcule la satisfacción de los pasajeros.

In [123]:

```
# Respuesta


def evaluate_time(q1, q2):
    time_per_pas = 30
    return max(len(q1), len(q2))*time_per_pas


def evaluate_sat(q1, q2, bq, pq, tq):

    num_pass = len(q1) + len(q2)

    # Tourists with children or reduced mobility
    t = sum([100 for p in q1 if p in tq])

    # Other tourists
    t2 = sum([25 for p in q2 if q2.index(p) < tq.index(p)])

    # First class
    p = sum([-50*(q1.index(p)-pq.index(p)) for p in q1
             if p in pq if q1.index(p) > pq.index(p)])

    return (t + t2 + p)/num_pass
```

7.5 Calculad el tiempo que se tarda en embarcar el vuelo 714 y la satisfacción de los clientes.

In [124]:

```
# Respuesta

q1, q2 = compute_new_queues_alg_1(bq, pq, tq)
print("Time: {}".format(evaluate_time(q1, q2)))
print("Satisfaction: {}".format(evaluate_sat(q1, q2, bq, pq, tq)))
```

```
Time: 930
Satisfaction: 6.0
```

5.- Bibliografía

5.1.- Bibliografía básica

Os recomendamos revisar la documentación oficial de las funciones y clases descritas en esta unidad, que encontraréis enlazadas en cada uno de los apartados que las describen.

Además, os recomendamos revisar la especificación completa del formato de `strftime y strptime` del módulo `datetime` en la documentación oficial del módulo (https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes). No hace falta que conozcáis todos los comodines que se usan para describir fechas y horas, pero una lectura de la documentación os proporcionará una visión global de qué tipo de expresiones se pueden construir.

De manera análoga, os recomendamos leer el *Regular Expression HOWTO* (https://docs.python.org/3/howto/regex.html#regex-howto) de la documentación oficial de Python.

5.2.- Bibliografia adicional - Ampliación de conocimientos

En esta unidad hemos presentado una pequeña introducción al uso de expresiones regulares con el módulo `re` (https://docs.python.org/3.8/library/re.html) de Python. Si queréis descubrir todos los detalles sobre el lenguaje de expresiones regulares y sobre las funciones que implementa el módulo `re` de Python, podéis dirigiros a la página oficial del módulo `re` (https://docs.python.org/3/library/re.html).

Por otro lado, los ejercicios resueltos de la unidad hacen uso de la función `copy` . Si queréis leer más sobre esta función y porqué es necesario su uso en el ejemplo, os recomendamos leer la documentación oficial del módulo `copy` (https://docs.python.org/3.8/library/copy.html) y esta pregunta (https://stackoverflow.com/questions/17246693/what-is-the-difference-between-shallow-copy-deepcopy-and-normal-assignment-oper) de stackoverflow. Intentad reflexionar sobre por qué, en el caso del ejemplo, hemos utilizado `copy` en vez de `deepcopy` , y si el código funcionaría correctamente usando `deepcopy` .