Programación para la ciencia de datos

Unidad 0: Afianzando conceptos

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.

Introducción

En esta unidad se repasan algunos de los conceptos básicos de programación Python que se han visto en la asignatura Fundamentos de Programación (FP), con el objetivo de afianzar los conceptos aprendidos y formalizarlos.

Si no habéis cursado la asignatura FP, se recomienda revisar los notebooks de aquella asignatura antes de seguir con el contenido de este notebook. Aunque aquí se explican los conceptos antes de formalizarlos, os aconsejamos revisar también la introducción más práctica que se proporciona en Fundamentos de programación, así como los ejemplos contextualizados en la ciencia de datos que allí se proporcionan.

Por otro lado, si ya habéis cursado la asignatura FP, este notebook debería servir, en primer lugar, como repaso de algunos de los conceptos más importantes de programación Python y, en segundo lugar, para reflexionar sobre estos conceptos y ver algunos detalles más formales que se pasaron por alto en la asignatura previa.

En este notebook se presentan, en primer lugar, las estructuras de datos básicos para almacenar colecciones en Python: las listas, las tuplas, los diccionarios y los conjuntos (revisad los notebooks de la asignatura FP si necesitáis repasar los tipos básicos de Python: enteros, flotantes, etc.).

Después, se explican dos conceptos que ya se han usado en FP, pero que no se han explicado formalmente hasta ahora: los iterables y la programación orientada a objetos.

A continuación, se revisan las instrucciones básicas de control de flujo de ejecución en Python: las estructuras alternativas, las iterativas, y las funciones.

Seguidamente, se presenta el PEP8, la guía de estilo de Python, y se explica cómo incorporar herramientas que nos ayuden a seguirla en los notebooks de jupyter.

Finalmente, se incluyen un conjunto de actividades para practicar, que permitirán poner en práctica lo explicado hasta ahora.

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

1.- Estructuras de datos para almacenar colecciones de valores

Python dispone de varias estructuras de datos que permiten almacenar colecciones de valores. Cada una de estas estructuras tiene unas propiedades diferentes y, por tanto, será útil para solucionar diferentes problemas.

1.1.- Listas

Una **lista** en Python es una colección **ordenada** de valores, posiblemente **heterogéneos**. Las listas se pueden modificar (son **mutables**), y permiten almacenar elementos **duplicados**.

Los elementos de una lista se encuentran ordenados, de tal manera que el primer elemento se encuentra indexado con el 0, el segundo con el 1, etc. El último elemento de la lista es pues indexado como el número de elementos de la lista menos uno.

In [82]:

```
# Las listas pueden ser heterogéneas y tener duplicados
a_list = [1, 1, 3.5, "strings also", [None, 4]]
print("The list is:\n\t{}".format(a_list))

# Las listas son mutables y ordenadas
a_list.append(5)
a_list.remove(3.5)
print("After appending 5 and removing 3.5:\n\t{}".format(a_list))
```

```
The list is:
        [1, 1, 3.5, 'strings also', [None, 4]]
After appending 5 and removing 3.5:
        [1, 1, 'strings also', [None, 4], 5]
```

Hacemos ahora un repaso rápido a los métodos implementados para las listas (encontraréis el detalle de todos los métodos aquí (https://docs.python.org/3/tutorial/datastructures.html#more-on-lists)). Podemos añadir elementos al final de un lista con `append` o en cualquier posición con `insert`. También podemos eliminar

un elemento de una lista a partir de su posición con `pop` o bien a partir de su valor con `remove` (`remove` elimina el primer elemento de la lista que coincide con el valor especificado). Dos listas se pueden concatenar con `extend` o bien con el operador de suma `+`.

In [83]:

```
# Añadimos un elemento al final de la lista
a_list.append(41)
print("After appending 41:\n\t{}".format(a_list))

# Añadimos un elemento al inicio de la lista
a_list.insert(0, -1)
print("After inserting -1:\n\t{}".format(a_list))

# Eliminamos el primer elemento
a_list.pop(0)
print("After removing the first element:\n\t{}".format(a_list))

# Eliminamos el elemento "strings also"
a_list.remove("strings also")
print("After removing 'strings also':\n\t{}".format(a_list))

# Concatena la lista con ella misma
a_list = a_list + a_list
print("After duplicating the list:\n\t{}".format(a_list))
```

```
After appending 41:
        [1, 1, 'strings also', [None, 4], 5, 41]
After inserting -1:
        [-1, 1, 1, 'strings also', [None, 4], 5, 41]
After removing the first element:
        [1, 1, 'strings also', [None, 4], 5, 41]
After removing 'strings also':
        [1, 1, [None, 4], 5, 41]
After duplicating the list:
        [1, 1, [None, 4], 5, 41, 1, 1, [None, 4], 5, 41]
```

También podemos recuperar la posición del primer elemento de una lista que tiene un cierto valor con `index`, o bien contar el número de veces que aparece un determinado elemento con `count`.

In [84]:

```
# Recuperamos el índice de la primera aparición del valor 5
i = a_list.index(5)
print("First 5 is in position:\n\t{}".format(i))

# Contamos el número de veces que el valor 1 aparece en la lista
c = a_list.count(1)
print("The number of 1s is:\n\t{}".format(c))
```

```
First 5 is in position:
        3
The number of 1s is:
        4
```

1.1.1- List slicing

La técnica de *list slicing* nos permite acceder a subconjuntos de elementos de una lista de manera sencilla y compacta. La sintaxis completa de *list slicing* consta del nombre de la variable que contiene la lista, seguida de `[X:Y:Z]`, donde `X` representa el inicio del fragmento que queremos recuperar, `Y` el final del fragmento y `Z` el paso o granularidad del fragmento:

In [85]:

```
a_list = ["A", "B", "C", "D", "E", "F"]
print(a_list)

# Mostramos los elementos en las posiciones de 0 a 2, saltando de uno en uno
print(a_list[0:3:1])
# Mostramos los elementos en las posiciones de 2 a 4, saltando de uno en uno
print(a_list[2:5:1])
# Mostramos los elementos en las posiciones de 0 a 4, saltando de dos en dos
print(a_list[0:5:2])
```

```
['A', 'B', 'C', 'D', 'E', 'F']
['A', 'B', 'C']
['C', 'D', 'E']
['A', 'C', 'E']
```

Fijaos como el elemento inicial se incluye en el resultado, mientras que el elemento final no. Es decir, si indicamos que queremos los elementos `[0:3:1]`, se incluirán los elementos con índices 0, 1 y 2; y si indicamos `[2:5:1]` los elementos con índice 2, 3 y 4.

El tercer valor (`Z`) nos permite indicar la granularidad de la selección: en los dos primeros ejemplos hemos elegido todos los elementos en los intervalos especificados (paso 1). En cambio, en el tercer ejemplo se saltan los elementos de 2 en 2, por lo que se incluyen los elementos con índices 0, 2 y 4.

Python permite omitir algunos de los valores en la especificación del *slicing*. Así, si se omite el inicio o el final del fragmento, se interpreta que seleccionamos el inicio de la lista o el final de la lista. Del mismo modo, si omitimos el paso se interpreta que este es 1:

In [163]:

```
# Omitimos el paso: mostramos los elementos en las posiciones de 2 a 4,
# saltando de uno en uno
print(a_list[2:5])

# Omitimos el inicio y el paso: mostramos los elementos desde el inicio hasta
# la posición 2, saltando de uno en uno
print(a_list[:3])

# Omitimos el inicio y el final: mostramos todos los elementos, saltando de dos
# en dos
print(a_list[::2])
```

Como último apunte, es interesante notar que podemos utilizar valores negativos en los índices. Así, por ejemplo, si queremos recorrer una lista en orden inverso, podemos hacerlo usando un paso de `-1`:

In [87]:

```
print(a_list[::-1])
```

```
['F', 'E', 'D', 'C', 'B', 'A']
```

1.1.2- List comprehensions

Una de las funcionalidades más usadas de las listas son las ***list comprehensions***, que permiten crear listas con expresiones muy concisas. Con las *list comprehensions* se pueden crear nuevas listas a partir de una (o varias) listas originales, operando sobre los valores originales y/o filtrándolos. La sintaxis de una *list comprehension* consta de unos corchetes (que definen la lista), que contienen al menos una cláusula `for` y que pueden tener también cláusulas `if` . Veámoslo con algunos ejemplos:

In [88]:

```
nums = range(10)
print(list(nums))

# Sintaxis básica con un solo for
nums_plus_3 = [n + 3 for n in nums]
print(nums_plus_3)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

A partir de la lista original, que contiene números del 0 al 9, hemos creado una *list comprehension* que genera una nueva lista que contiene los números de la lista original sumando tres a cada valor. Fijaos cómo, con este tipo de expresiones, podemos obtener listas de la misma longitud que las listas originales, pero que contienen el resultado de aplicar una función a los valores originales. Veamos algunos ejemplos más:

In [164]:

```
# Creamos una lista con los cuadrados de la lista original
nums_squared = [n**2 for n in nums]
print(nums_squared)


# Creamos una lista con los valores (i + 1) / (i - 1) para cada i de la
# lista original
def long_exp(i):
    if i != 1:
        r = (i + 1) / (i - 1)
    else:
        r = 0
    return r


nums_f = [long_exp(n) for n in nums]
print(nums_f)

# Creamos una lista con cadenas de caracteres "Num: n" para cada n de la
# lista original
nums_str = ["Num: " + str(n) for n in nums]
print(nums_str)
```

Tened en cuenta que una *list comprehension* puede evaluar una función (en el segundo ejemplo de la celda anterior, se evalúa la función `long_exp` para cada valor de la lista `nums`). Ahora bien, ¿podríamos obtener el mismo resultado sin definir la función `long_exp` ? Para hacerlo, podríamos usar una expresión `if` en una sola línea para obtener el mismo resultado sin tener que definir la función `long_exp` :

In [90]:

```
nums_f2 = [(n+1)/(n-1) if n != 1 else 0 for n in nums]
```

Así pues, hemos utilizado la sintaxis compacta del `if`, reduciéndolo a una sola línea de código. Fijaos que hemos aprovechado que los siguientes dos bloques de código son equivalentes:

In [91]:

```
i = 5

# Bloque 1:
if i != 1:
    r = (i + 1) / (i - 1)
else:
    r = 0

# Bloque 2:
r = (i+1)/(i-1) if i != 1 else 0
```

Es importante notar también que la lista que hemos creado, `nums_f2` tiene tantos elementos como tenía la lista `nums` sobre la que hemos iterado.

Las *list comprehension* también pueden incluir **condicionales** que sirven para filtrar qué valores de la (o las) listas originales se consideran en la creación de la nueva lista. En estos casos, la lista resultante tendrá una longitud igual o menor a la de la lista original, en función del número de veces que se cumpla la condición del `if`:

In [165]:

```
# Creamos una nueva lista que contiene únicamente los valores pares de la
# lista nums
nums_even = [n for n in nums if not n % 2]
print("nums:\t\t{}".format(list(nums)))
print("nums_even:\t{}".format(nums_even))
print("nums has {} elements and nums_even has {} elements\n".
      format(len(nums), len(nums_even)))

# Creamos una nueva lista a partir de a_list que contiene solo elementos
# que son enteros
a_list_of_ints = [n for n in a_list if type(n) == int]
print("a_list:\t\t{}".format(a_list))
print("a_list_of_ints:\t{}".format(a_list_of_ints))
print("a_list has {} elements and a_list_of_ints has {} elements\n".
      format(len(a_list), len(a_list_of_ints)))

# Creamos una nueva lista a partir de una lista de números escritos en letras
# que contiene los números tales que su expresión requiere más letras que el
# propio número
num_words = ["zero", "one", "two", "three", "four", "five", "six",
             "seven"]
num_words_c = [w for i, w in enumerate(num_words) if len(w) > i]
print("num_words:\t{}".format(num_words))
print("num_words_c:\t{}".format(num_words_c))
print("a_list has {} elements and num_words_c has {} elements".
      format(len(num_words), len(num_words_c)))
```

En el último ejemplo de la celda anterior, hemos utilizado `enumerate` en combinación con la *list comprehension* para generar el resultado deseado. La función `enumerate` aplicada sobre una lista devuelve otra lista de la misma longitud con tuplas de dos valores: el primer valor es el índice del elemento en la lista original, y el segundo valor es el elemento en cuestión:

In [93]:

```
# Observemos el resultado de enumerate
list(enumerate(['one', 'two', 'three']))
```

Out[93]:

```
[(0, 'one'), (1, 'two'), (2, 'three')]
```

Así, en cada iteración, la variable `i` contenía el índice del elemento y la variable `w` contenía el elemento. La condición que hemos aplicado era cierta si el número de letras de la palabra (`len(w)`) era superior al índice (`i`).

Las *list comprehension* no se limitan a una sola expresión `for` . Podemos utilizar más de una para construir una lista a partir de valores de varios iterables:

In [94]:

```
list_1 = [1, 2, 3]
list_2 = [10, 100]

# Creamos una lista de tuplas con las parejas de valores
# de list_1 y list_2
list_pairs = [(l1, l2) for l1 in list_1 for l2 in list_2]
print(list_pairs)

# Creamos una lista sumando las posibles combinaciones de valores
# de list_1 y list_2
lists_sum = [l1 + l2 for l1 in list_1 for l2 in list_2]
print(lists_sum)

# Creamos una lista de todas las palabras de 3 letras que se pueden hacer
# con las letras A, B y C
abc = ["A", "B", "C"]
lists_let = [l1 + l2 + l3
             for l1 in abc for l2 in abc for l3 in abc]
print(lists_let)
```

```
[(1, 10), (1, 100), (2, 10), (2, 100), (3, 10), (3, 100)]
[11, 101, 12, 102, 13, 103]
['AAA', 'AAB', 'AAC', 'ABA', 'ABB', 'ABC', 'ACA', 'ACB', 'ACC', 'BAA',
'BAB', 'BAC', 'BBA', 'BBB', 'BBC', 'BCA', 'BCB', 'BCC', 'CAA', 'CAB',
'CAC', 'CBA', 'CBB', 'CBC', 'CCA', 'CCB', 'CCC']
```

Finalmente, veamos un último ejemplo que combina tanto múltiples `for` como condiciones en una *list comprehension*:

In [95]:

```
# Creamos una lista de todas las palabras de 3 letras que se pueden hacer
# con las letras A, B y C y donde la primera y última letras son diferentes
lists_let = [l1 + l2 + l3
             for l1 in abc for l2 in abc for l3 in abc if l1 != l3]
print(lists_let)
```

```
['AAB', 'AAC', 'ABB', 'ABC', 'ACB', 'ACC', 'BAA', 'BAC', 'BBA', 'BBC',
'BCA', 'BCC', 'CAA', 'CAB', 'CBA', 'CBB', 'CCA', 'CCB']
```

1.2.- Tuplas

Las **tuplas** en Python son colecciones también **ordenadas** de elementos, posiblemente **heterogéneos** y con valores **duplicados**. Ahora bien, a diferencia de las listas, las tuplas son **inmutables**. Esto implica que una vez definidas, no podremos añadir ni eliminar elementos de una tupla, ni tampoco modificarlos:

In [96]:

```
# Definimos una tupla usando paréntesis
# Las tuplas pueden ser heterogéneas y tener duplicados
a_tuple = (1, 1, 3.5, "strings also", [None, 4])
print("The tuple is:\n\t{}".format(a_tuple))

# También podemos omitir los paréntesis en la definición de una tupla
the_same_tuple = 1, 1, 3.5, "strings also", [None, 4]
print("The tuple is:\n\t{}".format(the_same_tuple))

print("Are they equal:\n\t{}".format(a_tuple == the_same_tuple))

# Las tuplas son colecciones ordenadas:
print("First element is:\n\t{}".format(a_tuple[0]))
print("Second element is:\n\t{}".format(a_tuple[1]))
print("Third element is:\n\t{}".format(a_tuple[2]))
```

```
The tuple is:
        (1, 1, 3.5, 'strings also', [None, 4])
The tuple is:
        (1, 1, 3.5, 'strings also', [None, 4])
Are they equal:
        True
First element is:
        1
Second element is:
        1
Third element is:
        3.5
```

In [97]:

```
# Las tuplas son inmutables
try:
    del a_tuple[0]
except TypeError as e:
    print(e)

try:
    a_tuple[0] = "New value"
except TypeError as e:
    print(e)
```

```
'tuple' object doesn't support item deletion
'tuple' object does not support item assignment
```

1.3.- Diccionarios

Los **diccionarios** en Python son colecciones **sin orden** de elementos, posiblemente **heterogéneos** y **sin duplicados**.

Los **diccionarios** son la implementación, en Python, de la estructura de datos que conocemos con el nombre de *array asociativo* o *map*. Los diccionarios son colecciones de pares clave-valor, que además de las operaciones básicas de inserción, modificación y eliminación, también permiten recuperar los datos almacenados a través de la clave. La característica principal de esta estructura de datos es que no puede haber claves repetidas (cada clave aparece, como mucho, una única vez, y tiene por tanto un único valor asociado).

In [98]:

```
# Intentamos crear un diccionario con una clave repetida (a)
dict_0 = {"a": 0, "b": 1, "a": 2}
# Comprobamos como el diccionario tiene una única clave a:
dict_0
```

Out[98]:

```
{'a': 2, 'b': 1}
```

Ahora bien, un diccionario sí puede tener valores repetidos:

In [99]:

```
dict_0 = {"a": 0, "b": 0, "c": 0}
dict_0
```

Out[99]:

```
{'a': 0, 'b': 0, 'c': 0}
```

Hacemos ahora un repaso rápido a los métodos implementados para los diccionarios (encontraréis el detalle de todos los métodos aquí (https://docs.python.org/3.8/library/stdtypes.html#dict)). Podemos añadir elementos a un diccionario asignando el valor a la clave. Esta misma sintaxis sirve para actualizar los valores de un diccionario. También podemos eliminar un elemento de un diccionario a partir de su clave con `del` .

In [100]:

```
# Añadimos un elemento a dict_0
print("dict_0 is:\n\t{}".format(dict_0))
dict_0["d"] = 42
print("After adding d, dict_0 is:\n\t{}".format(dict_0))

# Actualizamos un elemento de dict_0
dict_0['a'] = -5
print("After updating a, dict_0 is:\n\t{}".format(dict_0))

# Eliminamos un elemento de dict_0
del dict_0['b']
print("After deleting b, dict_0 is:\n\t{}".format(dict_0))
```

```
dict_0 is:
        {'a': 0, 'b': 0, 'c': 0}
After adding d, dict_0 is:
        {'a': 0, 'b': 0, 'c': 0, 'd': 42}
After updating a, dict_0 is:
        {'a': -5, 'b': 0, 'c': 0, 'd': 42}
After deleting b, dict_0 is:
        {'a': -5, 'c': 0, 'd': 42}
```

Podemos recuperar todas las claves de un diccionario con el método `keys`, todos los valores con `values`, y ambos conjuntos de valores con `items`:

In [101]:

```
print("dict_0 is:\n\t{}".format(dict_0))

print("dict_0 keys are:\n\t{}".format(dict_0.keys()))

print("dict_0 values are:\n\t{}".format(dict_0.values()))

print("dict_0 items are:\n\t{}".format(dict_0.items()))
```

```
dict_0 is:
        {'a': -5, 'c': 0, 'd': 42}
dict_0 keys are:
        dict_keys(['a', 'c', 'd'])
dict_0 values are:
        dict_values([-5, 0, 42])
dict_0 items are:
        dict_items([('a', -5), ('c', 0), ('d', 42)])
```

Podemos iterar sobre los elementos de un diccionario utilizando `keys`, `values` o `items`, o bien iterando directamente sobre el diccionario (que es equivalente a iterar sobre sus claves):

In [167]:

```
# Veamos tres construcciones equivalentes que permiten mostrar
# las claves y los valores de un diccionario

# Opción 1: iteramos sobre el diccionario directamente
for k in dict_0:
    print("{}: {}".format(k, dict_0[k]))
print("\n")

# Opción 2: iteramos sobre las claves del diccionario
for k in dict_0.keys():
    print("{}: {}".format(k, dict_0[k]))
print("\n")

# Opción 3: iteramos sobre los items (pares de clave-valor)
for k, v in dict_0.items():
    # Tened en cuenta que aquí ya tenemos el elemento v, no es necesario
    # recuperarlo haciendo dict_0[k]
    print("{}: {}".format(k, v))
print("\n")

# También podemos iterar solo sobre los valores del diccionario
for value in dict_0.values():
    print(value)
```

Los diccionarios pueden contener otros diccionarios. Esto permite tener variables con estructuras complejas. Por ejemplo, imaginad que queremos guardar las velocidades máximas a las que se puede circular por autopista, carretera y ciudad en España y en Francia. Una alternativa sería usar un diccionario para guardar las velocidades de cada tipo de vía, y un segundo diccionario que guarde el diccionario de velocidades para cada país:

In [103]:

```
speeds = {
    "Spain": {"motorway": 120, "road": 90, "city": 50},
    "France": {"motorway": 130, "road": 80, "city": 50}
}

# Recuperamos el diccionario de las velocidades de España
print(speeds["Spain"])
# Consultamos la velocidad máxima en carretera en Francia
print(speeds["France"]["road"])
```

```
{'motorway': 120, 'road': 90, 'city': 50}
80
```

Los diccionarios de Python no tienen orden, es decir, los pares de clave-valor no se encuentran ordenados dentro de la estructura de datos. Así pues, dos diccionarios creados con pares de clave-valor iguales pero en órdenes diferentes, se consideran iguales:

In [104]:

```
# Creamos dos diccionarios con el mismo contenido, pero
# en orden diferente
dict_1 = {"a": 0, "b": 1, "c": 2}
dict_2 = {"b": 1, "a": 0, "c": 2}
# Comprobamos si los dos diccionarios son iguales
print(dict_1 == dict_2)
```

```
True
```

Ahora bien, a partir de la versión 3.6 de Python, la implementación de los diccionarios preserva el orden de inserción de los elementos. Es decir, cuando recorremos el diccionario, la implementación nos devuelve los elementos en el orden que fueron insertados. A partir de la versión 3.7 y posteriores, este comportamiento se ha declarado como oficial y, por lo tanto, podemos crear código que asuma que los diccionarios mantienen el orden de inserción de sus elementos:

In [105]:

```
# Creamos un diccionario
dict_3 = {"Key_" + str(num): "Value_" + str(num+1) for num in range(10)}

# Mostramos el contenido del diccionario en el orden en que items() devuelve
# los elementos
for k, v in dict_3.items():
    print("{}: {}".format(k, v))
```

```
Key_0: Value_1
Key_1: Value_2
Key_2: Value_3
Key_3: Value_4
Key_4: Value_5
Key_5: Value_6
Key_6: Value_7
Key_7: Value_8
Key_8: Value_9
Key_9: Value_10
```

In [106]:

```
# Borramos tres entradas del diccionario
del(dict_3["Key_5"])
del(dict_3["Key_8"])
del(dict_3["Key_1"])

# Añadimos dos nuevas entradas en el diccionario
dict_3["Key_10"] = "Value_10"
dict_3["Key_11"] = "Value_11"

# Modificamos el valor de Key_0
dict_3["Key_0"] = "New value does not change position"

# Finalmente, mostramos el contenido del diccionario siguiendo
# el orden devuelto por items()
for k, v in dict_3.items():
    print("{}: {}".format(k, v))
```

```
Key_0: New value does not change position
Key_2: Value_3
Key_3: Value_4
Key_4: Value_5
Key_6: Value_7
Key_7: Value_8
Key_9: Value_10
Key_10: Value_10
Key_11: Value_11
```

En el ejemplo anterior podemos ver cómo actualizar el valor de un elemento del diccionario (`Key_0`) no altera su orden (`Key_0` sigue siendo el primer elemento). Los elementos añadidos posteriormente (`Key_10` y `Key_11`) se muestran al final del diccionario, en el orden en que se han insertado.

1.3.1- *Dict comprehensions*

De una manera similar a las *list comprehensions* podemos utilizar *dict comprehensions* para crear nuevos diccionarios con una sintaxis compacta. La sintaxis de una *dict comprehension* consta de unas llaves (que definen el diccionario), que contienen al menos una cláusula `for` y que pueden tener también cláusulas `if` . Se deberá especificar cuál es la clave y cuál es el valor para cada entrada del diccionario (a diferencia de las listas, donde sólo había que especificar el valor de cada elemento). Veámoslo con algunos ejemplos:

In [168]:

```
# Definimos un diccionario sobre el que iterar
dict_4 = {1.0: "one", 2.0: "two", 3.0: "three", 4.0: "four", 5.0: "five"}
print("Original dict:\n\t{}".format(dict_4))

# Iteramos sobre las claves y creamos un nuevo diccionario con las mismas
# claves y "number" como valor (para todos los elementos)
dict_5 = {k: "number" for k in dict_4.keys()}
print("dict_5:\n\t{}".format(dict_5))

# Iteramos sobre los valores y creamos un nuevo diccionario utilizando
# los valores como clave y "new" como valor (para todos los elementos)
dict_6 = {v: "new" for v in dict_4.values()}
print("dict_6:\n\t{}".format(dict_6))

# Iteramos sobre los items y creamos un nuevo diccionario con las claves
# convertidas a entero y los valores con un ! final
dict_7 = {int(k): v + "!" for (k, v) in dict_4.items()}
print("dict_7:\n\t{}".format(dict_7))
```

A partir del diccionario original, hemos creado tres diccionarios nuevos utilizando *dict comprehensions*:

- Hemos creado el diccionario `dict_5` iterando sobre las claves del diccionario original. El diccionario `dict_5` contiene las mismas claves que el diccionario original, pero todos los valores asociados a estas claves son iguales (la cadena de caracteres `'number'`).
- Hemos creado el diccionario `dict_6` iterando sobre los valores del diccionario original. El diccionario `dict_6` tiene como claves los valores del diccionario original, y como valor todas tienen la misma cadena de caracteres (`'new'`). Tened en cuenta que, en este caso, el diccionario resultante tiene el mismo tamaño que el diccionario original ya que los valores del diccionario original no se encontraban repetidos.
- Hemos creado el diccionario `dict_7` iterando sobre los items (pares de clave-valor) del diccionario original. El diccionario `dict_7` tiene como claves las mismas claves que el diccionario original pero convertidas a enteros, y como valor los mismos valores que el diccionario original, pero terminados en un signo de exclamación.

Fijaos cómo, con este tipo de expresiones, podemos obtener diccionarios de la misma longitud que los diccionarios originales, y podemos operar sobre las claves y los valores. Veamos algunos ejemplos más:

In [108]:

```
# Creamos un diccionario con las mismas claves que el diccionario original
# pero pasadas a entero, y como valor guardamos la longitud del valor
# original (es decir, el número de letras de la palabra)
dict_8 = {int(k): len(v) for (k, v) in dict_4.items()}
print("dict_8:\n\t{}".format(dict_8))

# Creamos un diccionario con las mismas claves que el diccionario original
# pero pasadas a entero, y como valor guardamos el número de veces que
# aparece la letra e en el valor
dict_9 = {int(k): v.count("e") for (k, v) in dict_4.items()}
print("dict_9:\n\t{}".format(dict_9))

# Creamos un diccionario con los valores del diccionario original en mayúsculas
# como clave, y como valor guardamos la longitud del valor original (es
# decir, el número de letras de la palabra)
dict_10 = {v.upper(): len(v) for (k, v) in dict_4.items()}
print("dict_10:\n\t{}".format(dict_10))
```

```
dict_8:
        {1: 3, 2: 3, 3: 5, 4: 4, 5: 4}
dict_9:
        {1: 1, 2: 0, 3: 2, 4: 0, 5: 1}
dict_10:
        {'ONE': 3, 'TWO': 3, 'THREE': 5, 'FOUR': 4, 'FIVE': 4}
```

Del mismo modo que en las *list comprehensions*, las *dict comprehensions* pueden incluir condicionales que permiten filtrar qué elementos del diccionario original queremos considerar en la creación del nuevo diccionario. Así, el diccionario resultante tendrá una longitud igual o menor a la del diccionario original:

In [169]:

```
# Creamos un diccionario con las mismas claves que el diccionario original
# pero pasadas a entero, y los mismos valores, incluyendo solo los elementos
# que tienen alguna e en el valor
dict_11 = {int(k): v for (k, v) in dict_4.items() if v.count("e")}
print("dict_11:\n\t{}".format(dict_11))

# Creamos un diccionario que tiene como clave las claves originales pasadas a
# entero y sumando 10, y como valor el mismo valor concatenado con " + ten",
# incluyendo solo las claves impares
dict_12 = {int(k) + 10: v + " + ten"
           for (k, v) in dict_4.items() if int(k) % 2}
print("dict_12:\n\t{}".format(dict_12))
```

Por último, las *dict comprehensions* también pueden contener más de una cláusula `for`, lo que permite combinar los contenidos de varios diccionarios en la construcción del nuevo diccionario. Veamos un ejemplo de una baraja de cartas:

In [110]:

```
# Definimos los 4 palos y el símbolo que los representa
suits = {"hearts": "\u2665", "tiles": "\u2666",
         "clovers": "\u2663", "pikes": "\u2660"}
# Definimos los posibles valores de las cartas
ranks = {"2": 2, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7,
         "8": 8, "9": 9, "10": 10, "J": 11, "Q": 12, "K": 13, "A": 14}
# Definimos una posible asignación de valores a los palos
suit_cod = {"hearts": 1, "tiles": 2, "clovers": 3, "pikes": 4}
```

In [111]:

```
# Creamos un diccionario que contendrá todas las cartas de la baraja, con
# el símbolo del palo y el número de carta como clave, y el número de carta
# como valor
card_deck = {r_k + s_v: r_v
             for (s_k, s_v) in suits.items() for (r_k, r_v) in ranks.items()}
print(card_deck)
```

```
{'2♥': 2, '3♥': 3, '4♥': 4, '5♥': 5, '6♥': 6, '7♥': 7, '8♥': 8, '9♥':
9, '10♥': 10, 'J♥': 11, 'Q♥': 12, 'K♥': 13, 'A♥': 14, '2♦': 2, '3♦':
3, '4♦': 4, '5♦': 5, '6♦': 6, '7♦': 7, '8♦': 8, '9♦': 9, '10♦': 10, 'J
♦': 11, 'Q♦': 12, 'K♦': 13, 'A♦': 14, '2♣': 2, '3♣': 3, '4♣': 4, '5♣':
5, '6♣': 6, '7♣': 7, '8♣': 8, '9♣': 9, '10♣': 10, 'J♣': 11, 'Q♣': 12,
'K♣': 13, 'A♣': 14, '2♠': 2, '3♠': 3, '4♠': 4, '5♠': 5, '6♠': 6, '7♠':
7, '8♠': 8, '9♠': 9, '10♠': 10, 'J♠': 11, 'Q♠': 12, 'K♠': 13, 'A♠': 1
4}
```

In [112]:

```
# Creamos un diccionario que contendrá todas las cartas de la baraja, con
# el símbolo del palo y el número de carta como clave, y una codificación
# única como valor
card_deck_cod = {r_k + s_v: 100 * suit_cod[s_k] + r_v
                 for (s_k, s_v) in suits.items()
                 for (r_k, r_v) in ranks.items()}
print(card_deck_cod)
```

```
{'2♥': 102, '3♥': 103, '4♥': 104, '5♥': 105, '6♥': 106, '7♥': 107, '8
♥': 108, '9♥': 109, '10♥': 110, 'J♥': 111, 'Q♥': 112, 'K♥': 113, 'A♥':
114, '2♦': 202, '3♦': 203, '4♦': 204, '5♦': 205, '6♦': 206, '7♦': 207,
'8♦': 208, '9♦': 209, '10♦': 210, 'J♦': 211, 'Q♦': 212, 'K♦': 213, 'A
♦': 214, '2♣': 302, '3♣': 303, '4♣': 304, '5♣': 305, '6♣': 306, '7♣':
307, '8♣': 308, '9♣': 309, '10♣': 310, 'J♣': 311, 'Q♣': 312, 'K♣': 31
3, 'A♣': 314, '2♠': 402, '3♠': 403, '4♠': 404, '5♠': 405, '6♠': 406,
'7♠': 407, '8♠': 408, '9♠': 409, '10♠': 410, 'J♠': 411, 'Q♠': 412, 'K
♠': 413, 'A♠': 414}
```

1.4.- Conjuntos

Hay una cuarta estructura de datos que permite almacenar colecciones de valores en Python: los conjuntos. Los **conjuntos** en Python, como veremos en la próxima unidad, son colecciones **sin orden** de elementos, posiblemente **heterogéneos** y **sin duplicados**.

2.- Iterables

¿Qué tienen en común las tuplas, las listas y los diccionarios que hemos visto en el apartado anterior o, incluso, las cadenas de caracteres?

In [113]:

```
# Definimos una lista, una tupla, un diccionario y una cadena de caracteres
a_list = [1, 2, 3, 4]
a_tuple = (1, 2, 3, 4)
a_dict = {"one": 1, "two": 2, "three": 3, "four": 4}
a_str = "1234"

# Recorremos las estructuras con un for y mostramos su contenido
for e in a_list:
    print(e, end=" ")
print()

for e in a_tuple:
    print(e, end=" ")
print()

for e in a_dict:
    print(e, end=" ")
print()

for e in a_str:
    print(e, end=" ")
print()
```

```
1 2 3 4
1 2 3 4
one two three four
1 2 3 4
```

Todos ellos son objetos iterables en Python. Un objeto **iterable** es un objeto que implementa un método que devuelve un iterador sobre el objeto. Un **iterador** es un objeto que implementa el método `next`, que devuelve el siguiente elemento del contenedor iterable hasta que ya no quedan más elementos, momento en que lanza una excepción. Es decir, cuando estamos haciendo:

In [114]:

```
for e in a_list:
    print(e, end=" ")
```

```
1 2 3 4
```

internamente se está ejecutando un código similar al siguiente:

In [115]:

```
# Se crea un iterador de la lista
list_iterator = iter(a_list)
print(type(list_iterator))

# Se va llamando el método next() del iterador hasta que se produce
# una excepción de tipo StopIteration
while True:
    try:
        e = next(list_iterator)
        print(e, end=" ")
    except StopIteration:
        break
```

```
<class 'list_iterator'>
1 2 3 4
```

Hay varias funciones en Python que operan sobre iterables y que nos pueden ser útiles a la hora de procesar datos.

La función `zip` (https://docs.python.org/3.8/library/functions.html#zip) actúa sobre un conjunto de iterables, devolviendo un iterador de tuplas, donde la tupla número *i* contiene los elementos en la posición *i* de cada uno de los iterables:

In [116]:

```
# Definimos dos listas
nums = [1, 2, 3]
strs = ["one", "two", "three"]

# Usamos zip para crear una lista con tuplas de dos elementos,
# uno de cada una de las listas
nums_and_strs = zip(nums, strs)
print(list(nums_and_strs))
```

```
[(1, 'one'), (2, 'two'), (3, 'three')]
```

Esta función puede recibir cualquier número de iterables, y devolverá una lista de tuplas de tantos elementos como iterables ha recibido:

In [117]:

```
nums_floats = {1.0: 1, 2.0: 2, 3.0: 3, 4.0: 4, 5.0: 5}
nums_in_klingon = ("wa", "cha", "wej")

# Usamos zip para crear una lista con tuplas de cuatro elementos,
# uno de cada una de las listas
nums_and_strs = list(zip(nums, strs, nums_floats, nums_in_klingon))
print(nums_and_strs)
print("The result has {} elements".format(len(nums_and_strs)))
print("Each tuple has {} elements".format(len(nums_and_strs[0])))
print("The first tuple is: {}".format(nums_and_strs[0]))
```

```
[(1, 'one', 1.0, 'wa'), (2, 'two', 2.0, 'cha'), (3, 'three', 3.0, 'we
j')]
The result has 3 elements
Each tuple has 4 elements
The first tuple is: (1, 'one', 1.0, 'wa')
```

Es importante notar, por una parte, que `zip` trabaja con cualquier tipo de iterables (fijaos que en el último ejemplo combinamos el uso de listas, diccionarios y tuplas) y, por otra parte, que el resultado contendrá tantos elementos como elementos tenga el iterable más pequeño (tened en cuenta que, aunque la lista `nums_floats` tiene 5 elementos, el resultado solo tiene 3).

Ya hemos visto también la función `enumerate` (https://docs.python.org/3.8/library/functions.html#enumerate), que también actúa sobre iterables y devuelve un iterador de tuplas de dos elementos, donde la tupla número *i_ contiene el valor _i* (la posición del elemento dentro del iterable) y el elemento:

In [118]:

```
list(enumerate(nums_in_klingon))
```

Out[118]:

```
[(0, 'wa'), (1, 'cha'), (2, 'wej')]
```

Las funciones `any` (https://docs.python.org/3.8/library/functions.html#any) y `all` (https://docs.python.org/3.8/library/functions.html#all) actúan también sobre iterables y devuelven un booleano que nos indica si hay algún valor dentro del iterable que evalúa a True y si todos los valores dentro del iterable evalúan a True, respectivamente:

In [119]:

```
# Definimos una lista con todo True
a_list_of_trues = [True, True, True, True]
# Definimos una lista con algunos valores False y un True
a_list_with_a_true = [False, False, True, False]

print("Any on a list of trues:\t\t{}".format(any(a_list_of_trues)))
print("Any on a list with a True:\t{}".format(any(a_list_with_a_true)))

print("All on a list of trues:\t\t{}".format(all(a_list_of_trues)))
print("All on a list with a True:\t{}".format(all(a_list_with_a_true)))
```

```
Any on a list of trues:         True
Any on a list with a True:      True
All on a list of trues:         True
All on a list with a True:      False
```

Las funciones pueden trabajar sobre iterables que contienen elementos no booleanos (diferentes de `True` y `False`). En estos casos, su conversión a booleano se tiene en cuenta:

In [120]:

```
a_list_of_true_eqs = [True, 1, 2, 3, "something", 5.3]
a_list_of_false_eqs = [False, 0, 0.3, ""]

print("All on a list of elements that evaluate to True:\t{}".
      format(all(a_list_of_true_eqs)))
print("Any on a list of elements that evaluate to False:\t{}".
      format(all(a_list_of_false_eqs)))
```

```
All on a list of elements that evaluate to True:        True
Any on a list of elements that evaluate to False:       False
```

3.- Una pincelada de programación orientada a objetos

Habremos oído hablar y tendremos una intuición de que son las clases, los objetos y los métodos en Python. ¿Qué significan, sin embargo, estos conceptos? Aunque en otras asignaturas daremos un tratamiento más formal, aquí intentaremos presentar un resumen de los conceptos más básicos para empezar a entender cómo funciona Python.

Las clases y los objetos son conceptos básicos del paradigma de **programación orientada a objetos**. Python es un lenguaje orientado a objetos, así como Ruby, Scala, Java o C ++, entre otros.

Una **clase** es un prototipo, una plantilla, para crear **objetos** (instancias de la plantilla).

Así, por ejemplo, ya conocemos la clase `int` o la clase `float`, que nos permiten crear objetos con valores enteros o reales, respectivamente.

In [121]:

```
# Ya conocemos la clase int, que representa valores enteros
an_int = int(5)
# an_int es una instancia de la clase int
print(type(an_int))

# La clase int tiene un método bit_length que devuelve el número
# de bits necesarios para almacenar el entero
an_int.bit_length()
```

```
<class 'int'>
```

Out[121]:

```
3
```

In [122]:

```
# Ya conocemos la clase float, que representa valores reales
a_float = float(5.0)
# a_float es una instancia de la clase float
print(type(a_float))

# La clase float tiene un método is_integer que devuelve un booleano
# indicando si el valor almacenado es o no un entero
a_float.is_integer()
```

```
<class 'float'>
```

Out[122]:

```
True
```

Así, las variables `an_int` o `a_float` son instancias de las clases `int` y `float`. Las clases tienen definidos unos **métodos**, que podemos llamar con la sintaxis `instancia.nombre_del_método()`. Cada clase tiene definido un conjunto de métodos: por ejemplo, la clase `int` dispone del método `bit_length` y la clase `float` del método `is_integer`.

Más allá de utilizar clases ya definidas, Python también nos permite crear nuestras propias clases. Veamos un ejemplo: vamos a crear la clase `tea` que nos permitirá representar tés. De cada té, guardaremos el nombre, el tipo y el tiempo de infusión ideal. Esto serán **atributos** de cada instancia de la clase té, ya que cada instancia (cada té individual) tendrá sus propios valores de estas variables. De un té, querremos saber sus datos, lo querremos infusionar, y querremos saber si es compatible con otro té. Esto son comportamientos asociados al té, que implementaremos a través de métodos. Todos los tés se sirven en tazas (`cup`), independientemente del tipo de té que sean. Por lo tanto, la clase tiene tendrá un atributo de clase que informará de ello, y que será común a todas las instancias de la clase:

In [123]:

```
from time import sleep


class tea:

    # Definimos el atributo de clase recipient
    recipient = 'cup'

    def __init__(self, name, type_of_tea, brewing_time):
        # Definimos los atributos de instancia name, type_of_tea y brewing_time
        self.name = name
        self.type_of_tea = type_of_tea
        self.brewing_time = brewing_time

    # Definimos el método de instancia print_me
    def print_me(self):
        print("{} is a {} tea (brewing time {} seconds) served in a {}".format(
            self.name, self.type_of_tea, self.brewing_time, self.recipient))

    # Definimos el método de instancia brew
    def brew(self):
        sleep(self.brewing_time)

    # Definimos el método de instancia can_be_mixed
    def can_be_mixed(self, another_tea):
        return self.type_of_tea == another_tea.type_of_tea
```

In [170]:

```
# Creamos tres instancias de la clase té
morning_tea = tea("Earl grey", "Black", 4)
exotic_tea = tea("Chai late", "Black", 4)
jap_tea = tea("Sencha", "Green", 2)
jap_tea_2 = tea("Matcha", "Red", 3)

# Ejecutamos el método print_me de las tres instancias (observamos como cada
# de ellas contiene los atributos de instancia que hemos indicado al crear las
# instancias)
morning_tea.print_me()
exotic_tea.print_me()
jap_tea.print_me()
jap_tea_2.print_me()
print("\n")

# Hemos detectado un error en los atributos del té Matcha, y procedemos a
# modificarlos: fijaos como el cambio solo afecta a los atributos del
# té Matcha, y no a los atributos de ninguno de los otros tés
jap_tea_2.type_of_tea = "Green"
jap_tea_2.brewing_time = 1
morning_tea.print_me()
exotic_tea.print_me()
jap_tea.print_me()
jap_tea_2.print_me()

# Ejecutamos el método brew de jap_tea (esperemos 2 segundos)
jap_tea.brew()

# Ejecutamos el método can_be_mixed de morning_tea para comprobar si es
# compatible con exotic_tea
morning_tea.can_be_mixed(exotic_tea)
```

Out[170]:

```
True
```

Fijaos como este paradigma de programación permite **encapsular** código. Por ejemplo, el programador puede infusionar un té o comprobar si dos tés son compatibles sin saber cómo funcionan internamente estos métodos (no tiene porque saber el tiempo de infusionado de cada té o cuáles son las condiciones que deben cumplirse para que dos tés sean compatibles, simplemente llama a los métodos que ya incorporan el comportamiento).

Es interesante notar también que los métodos los llamamos utilizando la sintaxis `instancia.nombre_del_método()` , a excepción del método `__init__` . Este es un método especial que llamamos **constructor**, y que define cómo construimos las instancias de cada clase. En este caso, necesitamos tres parámetros, `name` , `type_of_tea` , y `brewing_time` , que especificaremos cuando construimos el objeto:

```
morning_tea = tea("Earl grey", "Black", 4)
exotic_tea = tea("Chai late", "Black", 4)
jap_tea = tea("Sencha", "Green", 2)
jap_tea_2 = tea("Matcha", "Red", 3)
```

Durante el curso no necesitaremos crear nuestras propias clases, pero sí utilizaremos las clases que las librerías de Python nos proporcionan para trabajar. Por lo tanto, no es imprescindible entender todos los detalles de cómo definimos clases, pero sí es importante entender qué es un objeto, como podemos llamar a los métodos que este implementa y cómo podemos acceder a sus atributos.

4.- Control de flujo de ejecución

Si no incluimos alguna instrucción que indique lo contrario, los programas en Python ejecutan las instrucciones secuencialmente, una tras otra (de arriba a abajo):

In [125]:

```
print("Pedestrian arrives to intersection")
print("Pedestrian crosses the street")
print("Pedestrian arrives to destination")
```

```
Pedestrian arrives to intersection
Pedestrian crosses the street
Pedestrian arrives to destination
```

Las tres instrucciones anteriores, que contienen un `print` de un mensaje, se ejecutan secuencialmente: el peatón llega a la intersección, luego cruza la calle y finalmente llega al destino.

Existen una serie de instrucciones que nos permiten alterar este flujo secuencial de los programas: las estructuras de control **alternativas** o condicionales (con `if-elif-else`) y las estructuras de control **iterativas** o bucles (con `for` o bien `while`).

4.1.- Estructuras de control alternativas

La instrucción `if` (https://docs.python.org/3.8/tutorial/controlflow.html#if-statements) nos permite ejecutar un bloque de código si se cumple una determinada condición. Si la condición no se cumple, se pueden comprobar condiciones adicionales con cláusulas `elif` o bien se puede ejecutar un segundo bloque de código especificado en la cláusula `else` . Las cláusulas `elif` y `else` son opcionales.

In [126]:

```
light_color = "green"

print("Pedestrian arrives to intersection")

if light_color == 'green':
    print("Pedestrian crosses the street")
    print("Pedestrian arrives to destination")
```

```
Pedestrian arrives to intersection
Pedestrian crosses the street
Pedestrian arrives to destination
```

En la celda de código anterior, el semáforo se encuentra en verde. El peatón llega al paso de peatones y, como el semáforo está en verde, se cumple la condición del `if ( light_color == 'green'`), por lo que el peatón cruza la calle y llega a su destino.

En cambio, en la celda siguiente el semáforo se encuentra en amarillo. Cuando el peatón llega al paso de peatones, la condición del `if` no se cumple, y el bloque de código dentro del `if` no se ejecuta.

In [127]:

```
light_color = "yellow"

print("Pedestrian arrives to intersection")

if light_color == 'green':
    print("Pedestrian crosses the street")
    print("Pedestrian arrives to destination")
```

```
Pedestrian arrives to intersection
```

Los dos ejemplos anteriores solo tenían una cláusula `if` . Podemos incluir también una cláusula `else` que especifique qué hacer si la condición no se cumple:

In [128]:

```
light_color = "green"

print("Pedestrian arrives to intersection")

if light_color == 'green':
    print("Pedestrian crosses the street")
    print("Pedestrian arrives to destination")
else:
    print("Pedestrian remains still")
```

```
Pedestrian arrives to intersection
Pedestrian crosses the street
Pedestrian arrives to destination
```

In [129]:

```
light_color = "yellow"

print("Pedestrian arrives to intersection")

if light_color == 'green':
    print("Pedestrian crosses the street")
    print("Pedestrian arrives to destination")
else:
    print("Pedestrian remains still")
```

```
Pedestrian arrives to intersection
Pedestrian remains still
```

También podemos incluir cláusulas `elif` con condiciones adicionales:

In [130]:

```
light_color = "green"

print("Pedestrian arrives to intersection")

if light_color == 'green':
    print("Pedestrian crosses the street")
    print("Pedestrian arrives to destination")
elif light_color == 'yellow':
    print("Pedestrian gets ready to cross")
else:
    print("Pedestrian remains still")
```

```
Pedestrian arrives to intersection
Pedestrian crosses the street
Pedestrian arrives to destination
```

In [131]:

```
light_color = "yellow"

print("Pedestrian arrives to intersection")

if light_color == 'green':
    print("Pedestrian crosses the street")
    print("Pedestrian arrives to destination")
elif light_color == 'yellow':
    print("Pedestrian gets ready to cross")
else:
    print("Pedestrian remains still")
```

```
Pedestrian arrives to intersection
Pedestrian gets ready to cross
```

In [132]:

```
light_color = "red"

print("Pedestrian arrives to intersection")

if light_color == 'green':
    print("Pedestrian crosses the street")
    print("Pedestrian arrives to destination")
elif light_color == 'yellow':
    print("Pedestrian gets ready to cross")
else:
    print("Pedestrian remains still")
```

```
Pedestrian arrives to intersection
Pedestrian remains still
```

Las condiciones de un `if` pueden ser tan complejas como sea necesario: cualquier expresión que evalúe a `True o False` se puede incluir en una cláusula `if` . Además, podemos incluir estructuras `if-elif-else` dentro de otras estructuras, a fin de generar flujos de ejecución complejos.

In [133]:

```
light_color = "green"
car_blocking_pass = True
pedestrian_distracted = False

print("Pedestrian arrives to intersection")

if light_color == 'green' and not pedestrian_distracted:
    if not car_blocking_pass:
        print("Pedestrian crosses the street")
        print("Pedestrian arrives to destination")
    else:
        print("Pedestrian yells!")
elif light_color == 'yellow' and not pedestrian_distracted:
    print("Pedestrian gets ready to cross")
else:
    print("Pedestrian remains still")
```

```
Pedestrian arrives to intersection
Pedestrian yells!
```

Os aconsejamos que modifiquéis los valores de las variables del código de la celda anterior y vayáis ejecutando el código, para comprobar cómo se comporta en cada situación.

4.2.- Estructuras de control iterativas

Las estructuras de control iterativas permiten ejecutar un mismo bloque de código varias veces.

La instrucción `while` (https://docs.python.org/3.8/reference/compound_stmts.html#the-while-statement) permite ejecutar un fragmento de código varias veces, mientras se cumpla una condición:

In [134]:

```
light_color = "red"
its_before_color_change = 10

print("Pedestrian arrives to intersection")

while light_color != "green":

    print("Pedestrian remains still")

    if light_color == 'yellow':
        print("Pedestrian gets ready to cross")
        light_color = "green"
    else:
        its_before_color_change = its_before_color_change - 1
        if its_before_color_change == 0:
            light_color = "yellow"

print("Pedestrian crosses the street")
print("Pedestrian arrives to destination")
```

```
Pedestrian arrives to intersection
Pedestrian remains still
Pedestrian remains still
Pedestrian remains still
Pedestrian remains still
Pedestrian remains still
Pedestrian remains still
Pedestrian remains still
Pedestrian remains still
Pedestrian remains still
Pedestrian remains still
Pedestrian remains still
Pedestrian gets ready to cross
Pedestrian crosses the street
Pedestrian arrives to destination
```

En el fragmento anterior, el código de dentro del `while` se ejecuta mientras el semáforo no sea verde. La lógica de condicionales dentro del `if` hace que el semáforo pase de rojo a amarillo después de 10 iteraciones, y que el semáforo pase de amarillo a verde a la siguiente iteración.

La instrucción `for` (https://docs.python.org/3.8/reference/compound_stmts.html#the-for-statement) también permite crear bucles, en este caso iterando sobre una secuencia de objetos:

In [135]:

```
for i in ["green", "yellow", "red"]:
    print("The color is now: {}".format(i))
```

```
The color is now: green
The color is now: yellow
The color is now: red
```

En cada iteración del bucle, la variable `i` toma el valor de uno de los elementos de la lista, en este caso, de uno de los posibles colores del semáforo.

En Python es muy habitual utilizar bucles `for` combinados con `range` (https://docs.python.org/3.8/library/stdtypes.html#range), una función que crea rangos de números:

In [136]:

```
for i in range(10):
    print("The number is {}".format(i))
```

```
The number is 0
The number is 1
The number is 2
The number is 3
The number is 4
The number is 5
The number is 6
The number is 7
The number is 8
The number is 9
```

Podemos incluir estructuras iterativas dentro de otras estructuras iterativas, para codificar el flujo que necesitamos para nuestro programa. Por ejemplo, vamos a generar todas las cartas de una baraja francesa:

In [137]:

```
suits = ["\u2665", "\u2666", "\u2663", "\u2660"]

playing_cards = []

for s in suits:

    for r in range(2, 15):

        if r == 14:
            t = "A"
        elif r == 13:
            t = "K"
        elif r == 12:
            t = "Q"
        elif r == 11:
            t = "J"
        else:
            t = str(r)

        playing_cards.append(t + s)

print(playing_cards)
```

```
['2♥', '3♥', '4♥', '5♥', '6♥', '7♥', '8♥', '9♥', '10♥', 'J♥', 'Q♥', 'K
♥', 'A♥', '2♦', '3♦', '4♦', '5♦', '6♦', '7♦', '8♦', '9♦', '10♦', 'J♦',
'Q♦', 'K♦', 'A♦', '2♣', '3♣', '4♣', '5♣', '6♣', '7♣', '8♣', '9♣', '10
♣', 'J♣', 'Q♣', 'K♣', 'A♣', '2♠', '3♠', '4♠', '5♠', '6♠', '7♠', '8♠',
'9♠', '10♠', 'J♠', 'Q♠', 'K♠', 'A♠']
```

La instrucción `break` (https://docs.python.org/3.8/reference/simple_stmts.html#break) permite salir de un bucle en un momento dado, deteniendo su su ejecución. Por ejemplo, el siguiente bucle `for` debería ejecutarse 10 veces si consideramos el número de elementos de `range(10)` , pero a la quinta iteración ($i = 4$) ejecuta el `break` y se interrumpe la ejecución del bucle:

In [138]:

```
for i in range(10):
    print("We are in iteration {}".format(i))
    if i == 4:
        break
```

```
We are in iteration 0
We are in iteration 1
We are in iteration 2
We are in iteration 3
We are in iteration 4
```

4.3.- Funciones

El uso de funciones también altera el flujo de ejecución lineal de un programa. Así, cuando definimos una función (indicada con la palabra clave `def`) el código dentro de la función no se ejecuta; en cambio, cuando llamamos a la función, ejecutaremos el código que esta contiene:

In [139]:

```
# Definimos la función suma
def suma(a, b):
    # El cuerpo de la función no se ejecuta en el momento de la definición
    r = a + b
    print("{} + {} = {}".format(a, b, r))
    return r


# Llamamos a la función suma: el cuerpo de la función se ejecutará a continuación
suma(3, 5)
suma(10, 2)
```

```
3 + 5 = 8
10 + 2 = 12
```

Out[139]:

```
12
```

Más adelante veremos con detalle las posibilidades que nos ofrece la definición de funciones y las diferentes opciones en su definición y uso. De momento, es importante recordar que nos permiten alterar el flujo lineal de ejecución de un programa, y que son clave en la creación de código modular (una propiedad importante para conseguir código claro y fácil de mantener).

5.- Guía de estilo

Una guía de estilo de código es un documento que describe un conjunto de reglas y recomendaciones a seguir cuando escribimos código en un lenguaje determinado. El PEP8 (https://www.python.org/dev/peps/pep-0008/) es la especificación que recoge la guía de estilo de Python. Por favor, **leed ahora este documento** e intentad seguir las indicaciones que se dan cuando programéis en Python.

La guía debe interpretarse como un conjunto de recomendaciones, a seguir siempre y cuando el sentido común no indique lo contrario. Es decir, no son normas estrictas a cumplir, y en ciertas circunstancias será preferible no hacer caso a algunas de las recomendaciones. En general, sin embargo, seguir las recomendaciones de la guía hará que el código que escribimos sea fácilmente legible, tanto para nosotros mismos como para otros desarrolladores.

Existen varias herramientas que ayudan a los desarrolladores a seguir las guías estilo. En los notebooks de la asignatura, utilizaremos `pycodestyle`, una herramienta que podemos activar en los notebooks para mostrar mensajes de alerta cuando el código que escribimos se salte las recomendaciones de la guía de estilo. En primer lugar, cargaremos la extensión `pycodestyle_magic`. Después, la activaremos o desactivaremos utilizando las instrucciones `% pycodestyle_on` y `% pycodestyle_off`:

In [140]:

```
%load_ext pycodestyle_magic
```

```
The pycodestyle_magic extension is already loaded. To reload it, use:
  %reload_ext pycodestyle_magic
```

In [141]:

```
%pycodestyle_on
```

In [142]:

```
# Código que sigue la guía de estilo: no genera alertas
print("Wrong style")
```

In [143]:

```
# Código que no sigue la guía de estilo: genera un mensaje de alerta
print("Wrong style" )
```

```
2:20: E202 whitespace before ')'
2:20: E202 whitespace before ')'
```

Tened en cuenta que las dos instrucciones relativas a la revisión del estilo de código empiezan por el carácter `%`. Estas instrucciones no son sentencias de Python, sino instrucciones especiales del kernel que se utiliza en los notebooks para proveer funcionalidades adicionales. Estas instrucciones se conocen en inglés como *magic commands*.

6.- 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. Calculad la suma de los primeros 100 números pares.

In [144]:

```
# Respuesta
```

2. La secuencia de Fibonnacci es una secuencia en la que cada término es la suma de los dos términos anteriores (la secuencia comienza con los valores 1 y 1).

   $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Calculad la suma de los 50 primeros valores de la secuencia.

In [145]:

```
# Respuesta
```

3. Se dice que un número es primo si solo es divisible por sí mismo y por 1. Generad una lista con los 100 primeros números primos.

In [146]:

```
# Respuesta
```

4. Calculad la suma de los cuadrados de los números naturales entre 100 y 200 (ambos valores incluidos).

In [147]:

```
# Respuesta
```

5. Generad todos los números de 3 dígitos que se pueden generar con los dígitos 3, 5 y 7.

In [148]:

```
# Respuesta
```

6. Contad y almacenad el número de veces que aparece cada palabra en el siguiente texto.

Forty-two is a pronic number and an abundante number; its prime factorization 2 · 3 · 7 makes it the second sphenic number and also the second of the form (2 · 3 · r).

In [149]:

```
# Respuesta
```

7. Generad todas las posibles subcadenas de 3 letras de la siguiente palabra: 'Electrodinamometro'.

In [150]:

```
# Respuesta
```

8. Cread una función que devuelva todas las posibles subcadenas de n letras de una palabra cualquiera. La función recibirá como parámetros el número de letras `n` y la palabra `word`, y devolverá una lista con las

subcadenas.

In [151]:

```
# Respuesta
```

6.1.- Soluciones a los ejercicios para practicar

1. Calculad la suma de los primeros 100 números pares.

In [152]:

```
# Opción 1: utilizando una list comprehension que filtra los
# valores pares (i % 2 == 0) y sumando los elementos de la lista con sum:
suma = sum([i for i in range(200) if i % 2 == 0])
print(suma)
```

In [153]:

```
# Opción 2: creando un bucle for con un if dentro, por lo que solo sumamos
# los valores pares (i % 2 == 0)
suma = 0
for i in range(200):
    if i % 2 == 0:
        suma = suma + i
print(suma)
```

2. La secuencia de Fibonnacci es una secuencia en la que cada término es la suma de los dos términos anteriores (la secuencia comienza con los valores 1 y 1).

$$1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Calculad la suma de los 50 primeros valores de la secuencia.

In [154]:

```
# Opción 1: creando una lista con los valores a sumar

# Creamos una lista con los 50 primeros valores
fib = [1, 1]
for i in range(48):
    fib.append(fib[-1] + fib[-2])
print(fib)

# Sumamos los valores de la secuencia
suma = sum(fib)
print(suma)
```

In [155]:

```
# Opción 2: sin crear una lista con los valores de la secuencia

suma = 2
# Guardamos los dos últimos valores de la secuencia en las variables l1 y l2
l1, l2 = 1, 1
for i in range(48):
    n = l1 + l2
    suma += n
    l2 = l1
    l1 = n

print(suma)
```

3. Se dice que un número es primo si solo es divisible por sí mismo y por 1. Generad una lista con los 100 primeros números primos.

In [156]:

```
primes = [2]
i = 3
while len(primes) != 100:
    is_prime = True

    # Si i es divisible por cualquiera de los primo que ya hemos identificado,
    # entonces i no es primo
    for p in primes:
        if i % p == 0:
            is_prime = False
            break
    # Si i es primo, lo añadimos a la lista de primos
    if is_prime:
        primes.append(i)
    i += 2

print(primes)
```

4. Calculad la suma de los cuadrados de los números naturales entre 100 y 200 (ambos valores incluidos).

In [157]:

```
# Usamos sum sobre una list comprehension que nos genera los valores
# de los cuadrados solicitados
sum([x**2 for x in range(100, 201)])
```

Out[157]:

```
2358350
```

5. Generad todos los números de 3 dígitos que se pueden generar con los dígitos 3, 5 y 7.

In [158]:

```
digits = [3, 5, 7]
# Creamos tres bucles for que recorren la lista de posibles dígitos
result = []
for d1 in digits:
    for d2 in digits:
        for d3 in digits:
            result.append(100*d1 + 10*d2 + d3)

print(result)
```

6. Contad y almacenad el número de veces que aparece cada palabra en el siguiente texto.

Forty-two is a pronic number and an abundant number; its prime factorization 2 · 3 · 7 makes it the second sphenic number and also the second of the form (2 · 3 · r).

In [171]:

```
text = "Forty-two is a pronic number and an abundant number; "
text += "its prime factorization 2 · 3 · 7 makes it the second sphenic "
text += "number and also the second of the form (2 · 3 · r)"

# Usamos un diccionario para almacenar el número de veces que
# sale cada palabra
word_counter = {}
for word in text.split(" "):
    if word in word_counter:
        # Si ya hemos encontrado la palabra anteriormente, incrementamos
        # el contador
        word_counter[word] += 1
    else:
        # Si es la primera vez que veamos la palabra, ponemos el contador a 1
        word_counter[word] = 1

print(word_counter)
```

7. Generad todas las posibles subcadenas de 3 letras de la siguiente palabra: 'Electrodinamometro'.

In [160]:

```
word = "Electrodinamometro"
words = []
# Extraemos la subcadena de 3 letras que comienza en cada posible
# posición inicial
for i in range(len(word)-2):
    words.append(word[i:i+3])
print(words)
```

8. Cread una función que devuelva todas las posibles subcadenas de n letras de una palabra cualquiera. La función recibirá como parámetros el número de letras `n` y la palabra `word`, y devolverá una lista con las subcadenas.

In [161]:

```
def substrings(n, word):
    words = []
    for i in range(len(word)-n+1):
        words.append(word[i:i+n])
    return words
```

In [162]:

```
substrings(4, word)
```

Out[162]:

```
['Elec',
 'lect',
 'ectr',
 'ctro',
 'trod',
 'rodi',
 'odin',
 'dina',
 'inam',
 'namo',
 'amom',
 'mome',
 'omet',
 'metr',
 'etro']
```