Programació per a la ciència de dades

Unitat 0: Refermant conceptes

Instruccions d'ús

Aquest document és un notebook interactiu que intercala explicacions més aviat teòriques de conceptes de programació amb fragments de codi executables. Per aprofitar els avantatges que aporta aquest format, us recomanem que, en primer lloc, llegiu les explicacions i el codi que us proporcionem. D'aquesta manera tindreu un primer contacte amb els conceptes que hi exposem. Ara bé, **la lectura és només el principi!** Una vegada hagueu llegit el contingut proporcionat, no oblideu executar el codi proporcionat i modificar-lo per crear-ne variants, que us permetin comprovar que heu entès la seva funcionalitat i explorar-ne els detalls d'implementació. Per últim, us recomanem també consultar la documentació enllaçada per explorar amb més profunditat les funcionalitats dels mòduls presentats.

Introducció

En aquesta unitat es repassen alguns dels conceptes bàsics de programació Python que s'han vist a l'assignatura Fonaments de Programació (FP), amb l'objectiu de refermar els conceptes apresos i formalitzar-los.

Si no heu cursat l'assignatura FP, us recomanem revisar els notebooks d'aquella assignatura abans de seguir amb el contingut d'aquest notebook. Tot i que aquí s'expliquen els conceptes abans de formalitzar-los, us aconsellem revisar també la introducció més pràctica que es proporciona a Fonaments de programació, així com els exemples contextualitzats en la ciència de dades que allí es proporcionen.

D'altra banda, si ja heu cursat l'assignatura FP, aquest notebook hauria de servir, en primer lloc, com a repàs d'alguns dels conceptes més importants de programació Python i, en segon lloc, per reflexionar sobre aquest conceptes i veure'n alguns detalls més formals que es van passar per alt a l'assignatura prèvia.

En aquest notebook es presenten, en primer lloc, les estructures de dades bàsiques per a emmagatzemar col·leccions en Python: les llistes, les tuples, els diccionaris i els conjunts (reviseu els notebooks de l'assignatura FP si us cal repassar els tipus bàsics de Python: enters, floats, etc.).

Després, s'expliquen dos conceptes que ja s'han fet servir a FP, però que no s'han explicat formalment fins ara: els iterables i la programació orientada a objectes.

A continuació, es revisen les instruccions bàsiques de control de flux d'execució en Python: les estructures alternatives, les iteratives, i les funcions.

Seguidament, es presenta el PEP8, la guia d'estil de Python, i s'explica com incorporar eines que ens ajudin a seguir-la en els notebooks de jupyter.

Finalment, s'inclouen un conjunt d'activitats per practicar, que us permetran posar en pràctica el que s'ha explicat fins ara.

A continuació s'inclou la taula de continguts, que podeu fer servir per a navegar pel document:

1. Estructures de dades per emmagatzemar col·leccions de valors

1.1. Llistes
1.1.1 List slicing
1.1.2 List comprehensions
1.2. Tuples
1.3. Diccionaris
1.4. Conjunts
2. Iterables
3. Una pinzellada de programació orientada a objectes
4. Control de flux d'execució
4.1. Estructures de control alternatives
4.2. Estructures de control iteratives
4.3. Funcions
5. Guia d'estil
6. Exercicis per practicar
6.1. Solucions als exercicis per practicar

1.- Estructures de dades per emmagatzemar col·leccions de valors

Python disposa de diverses estructures de dades que permeten emmagatzemar col·leccions de valors. Cadascuna d'aquestes estructures té unes propietats diferents i, per tant, serà útil per solucionar diferents problemes.

1.1.- Llistes

Una **llista** en Python és una col·lecció **ordenada** de valors, possiblement **heterogenis**. Les llistes es poden modificar (són **mutables**), i permeten emmagatzemar elements **duplicats**.

Els elements d'una llista es troben ordenats, de tal manera que el primer element es troba indexat amb el 0, el segon amb l'1, etc. L'últim element de la llista és doncs indexat com al nombre d'elements de la llista menys u.

In [1]:

```
# Les llistes poden ser heterogènies i tenir duplicats
a_list = [1, 1, 3.5, "strings also", [None, 4]]
print("The list is:\n\t{}".format(a_list))

# Les llistes són mutables i ordenades
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]
```

Fem ara un repàs ràpid als mètodes implementats per a les llistes (trobareu el detall de tots els mètodes aquí (https://docs.python.org/3/tutorial/datastructures.html#more-on-lists)). Podem afegir elements al final d'un llista amb `append` o en qualsevol posició amb `insert` . També podem eliminar un element d'una llista a partir de

la seva posició amb `pop` o bé a partir del seu valor amb `remove` (`remove` elimina el primer element de la llista que coincideix amb el valor especificat). Dues llistes es poden concatenar amb `extend` o bé amb l'operador de suma `+`.

In [2]:

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

# Afegim un element a l'inici de la llista
a_list.insert(0, -1)
print("After inserting -1:\n\t{}".format(a_list))

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

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

# Concatenem la llista amb ella mateixa
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]
```

També podem recuperar la posició del primer element d'una llista que té un cert valor amb `index`, o bé comptar el número de vegades que apareix un determinat element amb `count`.

In [3]:

```
# Recuperem l'índex de la primera aparició del valor 5
i = a_list.index(5)
print("First 5 is in position:\n\t{}".format(i))

# Comptem el número de vegades que el valor 1 apareix a la llista
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 tècnica de *list slicing* ens permet accedir a subconjunts d'elements d'una llista de manera senzilla i compacta. La sintaxis completa de *list slicing* consta del nom de la variable que conté la llista, seguida de `[X:Y:Z]`, on `X` representa l'inici del fragment que volem recuperar, `Y` el final del fragment i `Z` el pas o granularitat del fragment:

In [4]:

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

# Mostrem els elements en les posicions de 0 a 2, saltant d'un en un
print(a_list[0:3:1])
# Mostrem els elements en les posicions de 2 a 4, saltant d'un en un
print(a_list[2:5:1])
# Mostrem els elements en les posicions de 0 a 4, saltant 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']
```

Fixeu-vos com l'element inicial s'inclou en el resultat, mentre que l'element final no. És a dir, si indiquem que volem els elements `[0:3:1]`, s'inclouran els elements amb índexs 0, 1 i 2; i si indiquem `[2:5:1]` els elements amb índex 2, 3 i 4.

El tercer valor (`Z`) ens permet indicar la granularitat de la selecció: en els dos primers exemples hem triat tots els elements en els intervals especificats (pas 1). En canvi, en el tercer exemple salten els elements de 2 en 2, de manera que s'inclouen els elements amb índexs 0, 2 i 4.

Python permet ometre alguns dels valors en l'especificació de l'*slicing*. Així, si s'omet l'inici o el final del fragment, s'interpreta que seleccionem l'inici de la llista o el final de la llista. De la mateixa manera, si ometem el pas s'interpreta que aquest és 1:

In [5]:

```
# Ometem el pas: mostrem els elements en les posicions de 2 a 4,
# saltant d'un en un
print(a_list[2:5])

# Ometem l'inici i el pas: mostrem els elements des de l'inici fins a la
# posició 2, saltant d'un en un
print(a_list[:3])

# Ometem l'inici i el final: mostrem tots els elements, saltant de dos en dos
print(a_list[::2])
```

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

Com a últim apunt, és interessant notar que podem fer servir valors negatius en els índex. Així, per exemple, si volem recórrer una llista en ordre invers, podem fer-ho fent servir un pas de `-1`:

In [6]:

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

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

1.1.2- List comprehensions

Una de les funcionalitats més usades de les llistes són les ***list comprehensions***, que permeten crear llistes amb expressions molt concises. Amb les *list comprehensions* es poden crear noves llistes a partir d'una (o diverses) llistes originals, operant sobre els valors originals i/o filtrant-los. La sintaxis d'una *list comprehension* consta d'uns claudàtors (que defineixen la llista), que contenen com a mínim una clàusula `for` i que poden tenir també clàusules `if` . Vegem-ho amb alguns exemples:

In [7]:

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

# Sintaxi bàsica amb un sol 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 llista original, que conté números del 0 al 9, hem creat una *list comprehension* que genera una nova llista que conté els números de la llista original sumant tres a cada valor. Fixeu-vos com, amb aquest tipus d'expressions, podem obtenir llistes de la mateixa longitud que les llistes originals, però que contenen el resultat d'aplicar una funció als valors originals. Vegem-ne alguns exemples més:

In [8]:

```
# Creem una llista amb els quadrats de la llista original
nums_squared = [n**2 for n in nums]
print(nums_squared)


# Creem una llista amb els valors (i+1)/(i-1) per cada i de la llista 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)

# Creem una llista amb cadenes de caràcters "Num: x" per cada x de la
# llista original
nums_str = ["Num: " + str(n) for n in nums]
print(nums_str)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[-1.0, 0, 3.0, 2.0, 1.6666666666666667, 1.5, 1.4, 1.3333333333333333,
1.2857142857142858, 1.25]
['Num: 0', 'Num: 1', 'Num: 2', 'Num: 3', 'Num: 4', 'Num: 5', 'Num: 6',
'Num: 7', 'Num: 8', 'Num: 9']
```

Fixeu-vos que una *list comprehension* pot avaluar una funció (en el segon exemple de la cel·la anterior, s'avalua la funció `long_exp` per cada valor de la llista `nums`). Ara bé, podríem obtenir el mateix resultat sense definir la funció `long_exp`? Per fer-ho, podríem fer servir una expressió `if` en una sola línia per obtenir el mateix resultat sense haver de definir la funció `long_exp`:

In [9]:

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

Així doncs, hem fet servir la sintaxi compacta de l'`if`, reduint-lo a una sola línia de codi. Fixeu-vos que hem aprofitat que els següents dos blocs de codi són equivalents:

In [10]:

```
i = 5

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

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

És important notar també que la llista que hem creat, `nums_f2` té tants elements com tenia la llista `nums` sobre la qual hem iterat.

Les *list comprehension* també poden incloure **condicionals** que serveixen per filtrar quins valors de la (o les) llistes originals es consideren en la creació de la nova llista. En aquests casos, la llista resultant tindrà una longitud igual o menor a la de la llista original, en funció del número de vegades que es compleixi la condició de l' `if` :

In [11]:

```
# Creem una nova llista que conté únicament els valors parells de la
# llista 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)))

# Creem una nova llista a partir d'a_list que conté només elements
# que són enters
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)))

# Creem una nova llista a partir d'una llista de números escrits en lletres que
# conté els números tals que la seva expressió requereix més lletres que el
# propi 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)))
```

```
nums:           [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
nums_even:      [0, 2, 4, 6, 8]
nums has 10 elements and nums_even has 5 elements

a_list:         ['A', 'B', 'C', 'D', 'E', 'F']
a_list_of_ints: []
a_list has 6 elements and a_list_of_ints has 0 elements

num_words:      ['zero', 'one', 'two', 'three', 'four', 'five', 'six',
'seven']
num_words_c:    ['zero', 'one', 'two', 'three']
a_list has 8 elements and num_words_c has 4 elements
```

A l'últim exemple de la cel·la anterior, hem fet servir `enumerate` en combinació amb la *list comprehension* per tal de generar el resultat desitjat. La funció `enumerate` aplicada sobre una llista retorna una altra llista de la mateixa longitud amb tuples de dos valors: el primer valor és l'índex de l'element a la llista original, i el segon valor és l'element en qüestió:

In [12]:

```
# Observem el resultat d'enumerate
list(enumerate(['one', 'two', 'three']))
```

Out[12]:

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

Així, a cada iteració, la variable `i` contenia l'índex de l'element i la variable `w` contenia l'element. La condició que hem aplicat era certa si el número de lletres de la paraula (`len(w)`) era superior a l'índex (`i`).

Les *list comprehension* no es limiten a una sola expressió `for` . Podem fer-ne servir més d'una per construir una llista a partir de valors de diversos iterables:

In [13]:

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

# Creem una llista de tuples amb les parelles de valors
# de list_1 i list_2
list_pairs = [(l1, l2) for l1 in list_1 for l2 in list_2]
print(list_pairs)

# Creem una llista sumant les possibles combinacions de valors
# de list_1 i list_2
lists_sum = [l1 + l2 for l1 in list_1 for l2 in list_2]
print(lists_sum)

# Creem una llista de totes les paraules de 3 lletres que es poden fer
# amb les lletres A, B i 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']
```

Finalment, vegem un últim exemple que combina tant múltiples `for` com condicions en una *list comprehension*:

In [14]:

```
# Creem una llista de totes les paraules de 3 lletres que es poden fer
# amb les lletres A, B i C i on la primera i última lletres són diferents
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.- Tuples

Les **tuples** en Python són col·leccions també **ordenades** d'elements, possiblement **heterogenis** i amb valors **duplicats**. Ara bé, a diferència de les llistes, les tuples són **immutables**. Això implica que una vegada definides, no podrem afegir ni eliminar elements d'una tupla, ni tampoc modificar-los:

In [15]:

```
# Definim una tupla fent servir parèntesis
# Les tuples poden ser heterogènies i tenir duplicats
a_tuple = (1, 1, 3.5, "strings also", [None, 4])
print("The tuple is:\n\t{}".format(a_tuple))

# També podem ometre els parèntesis en la definició d'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))

# Les tuples són col·leccions ordenades:
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 [16]:

```
# Les tuples són immutables
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.- Diccionaris

Els **diccionaris** en Python són col·leccions **sense ordre** d'elements, possiblement **heterogenis** i **sense duplicats**.

Els **diccionaris** són la implementació, en Python, de l'estructura de dades que coneixem amb el nom d'*array associatiu* o *map*. Els diccionaris són col·leccions de parells clau-valor, que a més de les operacions bàsiques d'inserció, modificació i eliminació, també permeten recuperar les dades emmagatzemades a través de la clau. La característica principal d'aquesta estructura de dades és que no hi pot haver claus repetides (cada clau apareix, com a molt, una única vegada, i té per tant un únic valor associat).

In [17]:

```
# Intentem crear un diccionari amb una clau repetida (a)
dict_0 = {"a": 0, "b": 1, "a": 2}
# Comprovem com el diccionari té una única clau a:
dict_0
```

Out[17]:

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

Ara bé, un diccionari sí que pot tenir valors repetits:

In [18]:

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

Out[18]:

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

Fem ara un repàs ràpid als mètodes implementats per als diccionaris (trobareu el detall de tots els mètodes aquí (https://docs.python.org/3.8/library/stdtypes.html#dict)). Podem afegir elements a un diccionari assignant el valor a la clau. Aquesta mateixa sintaxi serveix per actualitzar els valors d'un diccionari. També podem eliminar un element d'un diccionari a partir de la seva clau amb `del`.

In [19]:

```
# Afegim un element 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))

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

# Eliminem un element 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}
```

Podem recuperar totes les claus d'un diccionari amb el mètode `keys`, tots els valors amb `values`, i ambdós

conjunts de valors amb `items` :

In [20]:

```
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)])
```

Podem iterar sobre els elements d'un diccionari fent servir `keys`, `values` o `items`, o bé iterant directament sobre el diccionari (que és equivalent a iterar sobre les seves claus):

In [21]:

```
# Vegem tres construccions equivalents que permeten mostrar
# les claus i els valors d'un diccionari

# Opció 1: iterem sobre el diccionari directament
for k in dict_0:
    print("{}: {}".format(k, dict_0[k]))
print("\n")

# Opció 2: iterem sobre les claus del diccionari
for k in dict_0.keys():
    print("{}: {}".format(k, dict_0[k]))
print("\n")

# Opció 3: iterem sobre els items (parells de clau-valor)
for k, v in dict_0.items():
    # Fixeu-vos que aquí ja tenim l'element v, no cal recuperar-lo
    # fent dict_0[k]
    print("{}: {}".format(k, v))
print("\n")

# També podem iterar només sobre els valors del diccionari
for value in dict_0.values():
    print(value)
```

```
a: -5
c: 0
d: 42


a: -5
c: 0
d: 42


a: -5
c: 0
d: 42


-5
0
42
```

Els diccionaris poden contenir altres diccionaris. Això permet tenir variables amb estructures complexes. Per exemple, imagineu que volem desar les velocitats màximes a les quals es pot circular per autopista, carretera i ciutat a Espanya i a França. Una alternativa seria fer servir un diccionari per desar les velocitats de cada tipus de via, i un segon diccionari que desés el diccionari de velocitats per cada país:

In [22]:

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

# Recuperem el diccionari de les velocitats d'Espanya
print(speeds["Spain"])

# Consultem la velocitat màxima en carretera a França
print(speeds["France"]["road"])
```

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

Els diccionaris de Python no tenen ordre, és a dir, els parells de clau-valor no es troben ordenats dins l'estructura de dades. Així doncs, dos diccionaris creats amb parells de clau-valor iguals però en ordres diferents, es consideren iguals:

In [23]:

```
# Creem dos diccionaris amb el mateix contingut, però
# en ordre diferent
dict_1 = {"a": 0, "b": 1, "c": 2}
dict_2 = {"b": 1, "a": 0, "c": 2}
# Comprovem si els dos diccionaris són iguals
print(dict_1 == dict_2)
```

```
True
```

Ara bé, a partir de la versió 3.6 de Python, la implementació dels diccionaris preserva l'ordre d'inserció dels elements. És a dir, quan recorrem el diccionari, la implementació ens retorna els elements en l'ordre que van ser inserits. A partir de la versió 3.7 i posteriors, aquest comportament s'ha declarat com a oficial i, per tant, podem crear codi que assumeixi que els diccionaris mantenen l'ordre d'inserció dels seus elements:

In [24]:

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

# Mostrem el contingut del diccionari en l'ordre en què items() retorna
# els elements
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 [25]:

```
# Esborrem tres entrades del diccionari
del(dict_3["Key_5"])
del(dict_3["Key_8"])
del(dict_3["Key_1"])

# Afegim dues noves entrades al diccionari
dict_3["Key_10"] = "Value_10"
dict_3["Key_11"] = "Value_11"

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

# Finalment, mostrem el contingut del diccionari seguint
# l'ordre retornat per 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
```

A l'exemple anterior podem veure com actualitzar el valor d'un element del diccionari (`Key_0`) no n'altera el seu ordre (`Key_0` segueix sent el primer element). Els elements afegits posteriorment (`Key_10` i `Key_11`) es mostren al final del diccionari, en l'ordre en què s'han inserit.

1.3.1- *Dict comprehensions*

D'una manera similar a les *list comprehensions* podem fer servir *dict comprehensions* per a crear nous diccionaris amb una sintaxis compacta. La sintaxis d'una *dict comprehension* consta d'unes claus (que defineixen el diccionari), que contenen com a mínim una clàusula for i que poden tenir també clàusules if. Caldrà especificar quina és la clau i quin és el valor per a cada entrada del diccionari (a diferència de les llistes, on només calia especificar el valor de cada element). Vegem-ho amb alguns exemples:

In [26]:

```
# Definim un diccionari sobre el que iterarem
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))

# Iterem sobre les claus i creem un nou diccionari amb les mateixes claus
# i "number" com a valor (per a tots els elements)
dict_5 = {k: "number" for k in dict_4.keys()}
print("dict_5:\n\t{}".format(dict_5))

# Iterem sobre els valors i creem un nou diccionari fent servir
# els valors com a clau i "new" com a valor (per a tots els elements)
dict_6 = {v: "new" for v in dict_4.values()}
print("dict_6:\n\t{}".format(dict_6))

# Iterem sobre els items i creem un nou diccionari amb les claus
# convertides a enter i els valors amb un ! final
dict_7 = {int(k): v + "!" for (k, v) in dict_4.items()}
print("dict_7:\n\t{}".format(dict_7))
```

```
Original dict:
        {1.0: 'one', 2.0: 'two', 3.0: 'three', 4.0: 'four', 5.0: 'fiv
e'}
dict_5:
        {1.0: 'number', 2.0: 'number', 3.0: 'number', 4.0: 'number',
5.0: 'number'}
dict_6:
        {'one': 'new', 'two': 'new', 'three': 'new', 'four': 'new', 'f
ive': 'new'}
dict_7:
        {1: 'one!', 2: 'two!', 3: 'three!', 4: 'four!', 5: 'five!'}
```

A partir del diccionari original, hem creat tres diccionaris nous fent servir *dict comprehensions*:

- Hem creat el diccionari `dict_5` iterant sobre les claus del diccionari original. El diccionari `dict_5` conté les mateixes claus que el diccionari original, però tots els valors associats a aquestes claus són iguals (la cadena de caràcters `'number'`).
- Hem creat el diccionari `dict_6` iterant sobre els valors del diccionari original. El diccionari `dict_6` té com a claus els valors del diccionari original, i com a valor totes tenen la mateixa cadena de caràcters (`'new'`). Fixeu-vos que, en aquest cas, el diccionari resultant té la mateixa mida que el diccionari original ja que els valors del diccionari original no es trobaven repetits.
- Hem creat el diccionari `dict_7` iterant sobre els items (parells de clau-valor) del diccionari original. El diccionari `dict_7` té com a claus les mateixes claus que el diccionari original però convertides a enters, i com a valor els mateixos valors que el diccionari original, però acabats en un signe d'exclamació.

Fixeu-vos com, amb aquest tipus d'expressions, podem obtenir diccionaris de la mateixa longitud que els diccionaris originals, i podem operar sobre les claus i els valors. Vegem-ne alguns exemples més:

In [27]:

```
# Creem un diccionari amb les mateixes claus que el diccionari original
# però passades a enter, i com a valor hi desem la longitud del valor
# original (és a dir, el número de lletres de la paraula)
dict_8 = {int(k): len(v) for (k, v) in dict_4.items()}
print("dict_8:\n\t{}".format(dict_8))

# Creem un diccionari amb les mateixes claus que el diccionari original
# però passades a enter, i com a valor hi desem el número de vegades que
# apareix la lletra e al valor
dict_9 = {int(k): v.count("e") for (k, v) in dict_4.items()}
print("dict_9:\n\t{}".format(dict_9))

# Creem un diccionari amb els valors del diccionari original en majúscules
# com a clau, i com a valor hi desem la longitud del valor original (és a
# dir, el número de lletres de la paraula)
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}
```

De la mateixa manera que en les *list comprehensions*, les *dict comprehensions* poden incloure condicionals que permeten filtrar quins elements del diccionari original volem considerar en la creació del nou diccionari. Així, el diccionari resultant tindrà una longitud igual o menor a la del diccionari original:

In [28]:

```
# Creem un diccionari amb les mateixes claus que el diccionari original
# però passades a enter, i els mateixos valors, incloent només els elements
# que tenen alguna e al 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))

# Creem un diccionari que té com a clau les claus originals en enter i
# sumant 10, i com a valor el mateix valor concatenat amb "+ ten",
# incloent només les claus senars
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))
```

```
dict_11:
        {1: 'one', 3: 'three', 5: 'five'}
dict_12:
        {11: 'one + ten', 13: 'three + ten', 15: 'five + ten'}
```

Per últim, les *dict comprehensions* també poden contenir més d'una clàusula `for`, el que permet combinar els continguts de diversos diccionaris en la construcció del nou diccionari. Vegem-ne un exemple d'una baralla de cartes:

In [29]:

```
# Definim els 4 pals i el símbol que els representa
suits = {"hearts": "\u2665", "tiles": "\u2666",
         "clovers": "\u2663", "pikes": "\u2660"}
# Definim els possibles valors de les cartes
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}
# Definim una possible assignació de valors als pals
suit_cod = {"hearts": 1, "tiles": 2, "clovers": 3, "pikes": 4}
```

In [30]:

```
# Creem un diccionari que contindrà totes les cartes de la baralla, amb
# el símbol del pal i el número de carta com a clau, i el número de carta
# com a 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 [31]:

```
# Creem un diccionari que contindrà totes les cartes de la baralla, amb
# el símbol del pal i el número de carta com a clau, i una codificació
# única com a 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.- Conjunts

Hi ha una quarta estructura de dades que permet emmagatzemar col·leccions de valors en Python: els conjunts. Els **conjunts** en Python, com veurem a la propera unitat, són col·leccions **sense ordre** d'elements, possiblement **heterogenis** i **sense duplicats**.

2.- Iterables

Què tenen en comú les tuples, les llistes i els diccionaris que hem vist a l'apartat anterior o, fins i tot, les cadenes de caràcters?

In [32]:

```
# Definim una llista, una tupla, un diccionari i una cadena de caràcters
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"

# Recorrem les estructures amb un for i en mostrem el seu contingut
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
```

Tots ells són objectes iterables en Python. Un objecte **iterable** és un objecte que implementa un mètode que retorna un iterador sobre l'objecte. Un **iterador** és un objecte que implementa el mètode `next`, que va retornant el següent element del contenidor iterable fins que ja no queden més elements, moment en què llança una excepció. És a dir, quan estem fent:

In [33]:

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

```
1 2 3 4
```

internament s'està executant un codi similar al següent:

In [34]:

```
# Es crea un iterador de la llista
list_iterator = iter(a_list)
print(type(list_iterator))

# Es va cridant el mètode next() de l'iterador fins que es produeix
# una excepció de tipus StopIteration
while True:
    try:
        e = next(list_iterator)
        print(e, end=" ")
    except StopIteration:
        break
```

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

Hi ha diverses funcions en Python que operen sobre iterables i que ens poden ser útils a l'hora de processar dades.

La funció `zip` (https://docs.python.org/3.8/library/functions.html#zip) actua sobre un conjunt d'iterables, retornant un iterador de tuples, on la tupla número *i* conté els elements en la posició *i* de cadascun dels iterables:

In [35]:

```
# Definim dues llistes
nums = [1, 2, 3]
strs = ["one", "two", "three"]

# Fem servir zip per crear una llista amb tuples de dos elements,
# un de cadascuna de les llistes
nums_and_strs = zip(nums, strs)
print(list(nums_and_strs))
```

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

Aquesta funció pot rebre qualsevol número d'iterables, i retornarà una llista de tuples de tants elements com iterables ha rebut:

In [36]:

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

# Fem servir zip per crear una llista amb tuples de quatre elements,
# un de cadascuna de les llistes
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')
```

És important notar, d'una banda, que `zip` treballa amb qualsevol tipus d'iterables (fixeu-vos que a l'últim exemple combinem l'ús de llistes, diccionaris i tuples) i, d'altra banda, que el resultat contindrà tants elements com elements tingui l'iterable més petit (fixeu-vos que, tot i que la llista `nums_floats` té 5 elements, el resultat només en té 3).

Ja hem vist també la funció `enumerate` (https://docs.python.org/3.8/library/functions.html#enumerate), que també actua sobre iterables i retorna un iterador de tuples de dos elements, on la tupla número *i_ conté el valor _i* (la posició de l'element dins de l'iterable) i l'element:

In [37]:

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

Out[37]:

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

Les funcions `any` (https://docs.python.org/3.8/library/functions.html#any) i `all` (https://docs.python.org/3.8/library/functions.html#all) actuen també sobre iterables i retornen un booleà que ens indica si hi ha algun valor dins de l'iterable que avalua a True i si tots els valors dins de l'iterable avaluen a True, respectivament:

In [38]:

```
# Definim una llista amb tot Trues
a_list_of_trues = [True, True, True, True]
# Definim una llista amb alguns valors False i 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
```

Les funcions poden treballar sobre iterables que contenen elements no booleans (diferents de `True` i `False`). En aquests casos, la seva conversió a booleà es té en compte:

In [39]:

```
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 pinzellada de programació orientada a objectes

Haurem sentit parlar i tindrem una intuïció de què són les classes, els objectes i els mètodes en Python. Què volen dir, però, aquests conceptes? Tot i que en altres assignatures en donarem un tractament més formal, aquí intentarem presentar un resum dels conceptes més bàsics per començar a entendre com funciona Python.

Les classes i els objectes són conceptes bàsics del paradigma de **programació orientada a objectes**. Python és un llenguatge orientat a objectes, així com Ruby, Scala, Java o C++, entre d'altres.

Una **classe** és un prototip, una plantilla, per crear **objectes** (instàncies de la plantilla).

Així, per exemple, ja coneixem la classe `int` o la classe `float`, que ens permeten crear objectes amb valors enters o reals, respectivament.

In [40]:

```
# Ja coneixem la classe int, que representa valors enters
an_int = int(5)
# an_int és una instància de la classe int
print(type(an_int))

# La classe int té un mètode bit_length que retorna el número
# de bits necessaris per emmagatzemar l'enter
an_int.bit_length()
```

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

Out[40]:

```
3
```

In [41]:

```
# Ja coneixem la classe float, que representa valors reals
a_float = float(5.0)
# a_float és una instància de la classe float
print(type(a_float))

# La classe float té un mètode is_integer que retorna un booleà
# indicant si el valor emmagatzemat és o no un enter
a_float.is_integer()
```

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

Out[41]:

```
True
```

Així, les variables `an_int` o `a_float` són instàncies de les classes `int` i `float`. Les classes tenen definits uns **mètodes**, que podem cridar amb la sintaxis `instància.nom_del_mètode()`. Cada classe té definit un conjunt de mètodes: per exemple, la classe `int` disposa del mètode `bit_length` i la classe `float` del mètode `is_integer`.

Més enllà de fer servir classes ja definides, Python també ens permet crear les nostres pròpies classes. Vegem-ne un exemple: crearem la classe `tea` que ens permetrà representar tes. De cada te, en desarem el nom, el tipus i el temps d'infusió ideal. Això seran atributs de cada instància de la classe te, ja que cada instància (cada te individual) tindrà els seus propis valors d'aquestes variables. D'un te, en voldrem saber les seves dades, el voldrem infusionar, i voldrem saber si és compatible amb un altre te. Això són comportaments associats al te, que implementarem a través de mètodes. Tots els tes se serveixen en tasses (`cup`), independentment del tipus de te que siguin. Per tant, la classe té tindrà un atribut de classe que informarà d'això, i que serà comú a totes les instàncies de la classe:

In [42]:

```
from time import sleep


class tea:

    # Definim l'atribut de classe recipient
    recipient = 'cup'

    def __init__(self, name, type_of_tea, brewing_time):
        # Definim els atributs d'instància name, type_of_tea i brewing_time
        self.name = name
        self.type_of_tea = type_of_tea
        self.brewing_time = brewing_time

    # Definim el mètode d'instància 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))

    # Definim el mètode d'instància brew
    def brew(self):
        sleep(self.brewing_time)

    # Definim el mètode d'instància can_be_mixed
    def can_be_mixed(self, another_tea):
        return self.type_of_tea == another_tea.type_of_tea
```

In [43]:

```
# Creem tres instàncies de la classe te
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)

# Executem el mètode print_me de les tres instàncies (observem com cadascuna
# d'elles conté els atributs d'instància que hem indicat al crear les
# instàncies)
morning_tea.print_me()
exotic_tea.print_me()
jap_tea.print_me()
jap_tea_2.print_me()
print("\n")

# Hem detectat un error en els atributs del te Matcha, i procedim a
# modificar-los: fixeu-vos com el canvi només afecta als atributs del
# te Matcha, i no pas als atributs de cap dels altres tes
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()

# Executem el mètode brew de jap_tea (esperem 2 segons)
jap_tea.brew()

# Executem el mètode can_be_mixed de morning_tea per comprovar si és compatible
# amb exotic_tea
morning_tea.can_be_mixed(exotic_tea)
```

```
Earl grey is a Black tea (brewing time 4 seconds) served in a cup
Chai late is a Black tea (brewing time 4 seconds) served in a cup
Sencha is a Green tea (brewing time 2 seconds) served in a cup
Matcha is a Red tea (brewing time 3 seconds) served in a cup


Earl grey is a Black tea (brewing time 4 seconds) served in a cup
Chai late is a Black tea (brewing time 4 seconds) served in a cup
Sencha is a Green tea (brewing time 2 seconds) served in a cup
Matcha is a Green tea (brewing time 1 seconds) served in a cup
```

Out[43]:

```
True
```

Fixeu-vos com aquest paradigma de programació permet **encapsular** codi. Per exemple, el programador pot infusionar un te o comprovar si dos tes són compatibles sense saber com funcionen internament aquests mètodes (no té perquè saber el temps d'infusionat de cada te o quines són les condicions que s'han de complir perquè dos tes siguin compatibles, només cal que cridi als mètodes que ja incorporen el comportament).

És interessant notar també que els mètodes els cridem fent servir la sintaxis `instància.nom_del_mètode()` , a excepció del mètode `__init__` . Aquest és un mètode especial que anomenem constructor, i que defineix com construïm les instàncies de cada classe. En aquest cas, necessitem tres paràmetres, `name` , `type_of_tea` , i `brewing_time` , que especificarem quan construïm l'objecte:

```
    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)
```

Durant el curs no necessitarem crear les nostres pròpies classes, però sí que farem servir les classes que les llibreries de Python ens proporcionen per a treballar. Per tant, no és imprescindible entendre tots els detalls de com definim classes, però sí que és important entendre què és un objecte, com podem cridar als mètodes que aquest implementa i com podem accedir als seus atributs.

4.- Control de flux d'execució

Si no incloem alguna instrucció que indiqui el contrari, els programes en Python executen les instruccions seqüencialment, una darrera de l'altra (de dalt a baix):

In [44]:

```
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
```

Les tres instruccions anteriors, que contenen un `print` d'un missatge, s'executen seqüencialment: el vianant arriba a la intersecció, després creua el carrer i finalment arriba a la destinació.

Existeixen però un seguit d'instruccions que ens permeten alterar aquest flux seqüencial dels programes: les estructures de control **alternatives** o condicionals (amb `if-elif-else`) i les estructures de control **iteratives** o bucles (amb `for` o bé `while`).

4.1.- Estructures de control alternatives

La instrucció `if` (https://docs.python.org/3.8/tutorial/controlflow.html#if-statements) ens permet executar un bloc de codi si es compleix una determinada condició. Si la condició no es compleix, es poden comprovar condicions addicionals amb clàusules `elif` o bé es pot executar un segon bloc de codi especificat a la clàusula `else` . Les clàusules `elif` i `else` són però opcionals.

In [45]:

```
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
```

A la cel·la de codi anterior, el semàfor es troba en verd. El vianant arriba al pas de vianants i, com que el

semàfor és verd, es compleix la condició de l' `if ( light_color == 'green' )`, de manera que el vianant creua el carrer i arriba a la seva destinació.

En canvi, a la cel·la següent el semàfor es troba en groc. Quan el vianant arriba al pas de vianants, la condició de l' `if` no es compleix, i el bloc de codi de dins de l' `if` no s'executa.

In [46]:

```
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
```

Els dos exemples anteriors només tenien una clàusula `if`. Podem incloure també una clàusula `else` que especifiqui què cal fer si la condició no es compleix:

In [47]:

```
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 [48]:

```
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
```

També podem incloure clàusules `elif` amb condicions addicionals:

In [49]:

```
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 [50]:

```
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 [51]:

```
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
```

Les condicions d'un `if` poden ser tant complexes com sigui necessari: qualsevol expressió que avaluï a `True` o `False` es pot incloure en una clàusula `if` . A més, podem incloure estructures `if-elif-else` dins d'altres estructures, per tal de generar fluxos d'execució complexos.

In [52]:

```
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!
```

Us aconsellem que varieu els valors de les variables del codi de la cel·la anterior i aneu executant el codi, per comprovar com es comporta en cada situació.

4.2.- Estructures de control iteratives

Les estructures de control iteratives permeten executar un mateix bloc de codi diverses vegades.

La instrucció `while` (https://docs.python.org/3.8/reference/compound_stmts.html#the-while-statement) permet executar un fragment de codi diverses vegades, mentre es compleixi una condició:

In [53]:

```
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
```

Al fragment anterior, el codi de dins del `while` s'executa mentre el semàfor no sigui verd. La lògica de condicionals dins de l' `if` fa que el semàfor passi de vermell a groc després de 10 iteracions, i que el semàfor passi de groc a verd a la següent iteració.

La instrucció `for` (https://docs.python.org/3.8/reference/compound_stmts.html#the-for-statement) també permet crear bucles, en aquest cas iterant sobre una seqüència d'objectes:

In [54]:

```
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
```

A cada iteració del bucle, la variable `i` pren el valor d'un dels elements de la llista, en aquest cas, d'un dels possibles colors del semàfor.

En Python és molt habitual fer servir bucles `for` combinats amb `range` (https://docs.python.org/3.8/library/stdtypes.html#range), una funció que crea rangs de números:

In [55]:

```
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
```

Podem incloure estructures iteratives dins d'altres estructures iteratives, per tal de codificar el flux que necessitem per al nostre programa. Per exemple, anem a generar totes les cartes d'una baralla francesa:

In [56]:

```
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 instrucció `break` (https://docs.python.org/3/reference/simple_stmts.html#break) permet sortir d'un bucle en un moment donat, aturant-ne la seva execució. Per exemple, el següent bucle for hauria d'executar-se 10 vegades si considerem el número d'elements de `range(10)`, però a la cinquena iteració ($i = 4$) s'executa el break i s'interromp l'execució del bucle:

In [57]:

```
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.- Funcions

L'ús de funcions també altera el flux d'execució lineal d'un programa. Així, quan definim una funció (indicada amb la paraula clau `def`) el codi de dins de la funció no s'executa; en canvi, quan cridem la funció, executarem el codi que aquesta conté:

In [58]:

```
# Definim la funció suma
def suma(a, b):
    # El cos de la funció no s'executa en el moment de la definició
    r = a + b
    print("{} + {} = {}".format(a, b, r))
    return r


# Cridem la funció suma: el cos de la funció s'executarà a continuació
suma(3, 5)
suma(10, 2)
```

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

Out[58]:

```
12
```

Més endavant veurem amb detall les possibilitats que ens ofereix la definició de funcions i les diferents opcions en la seva definició i ús. De moment, és important recordar que ens permeten alterar el flux lineal d'execució d'un programa, i que són clau en la creació de codi modular (una propietat important per aconseguir codi clar i fàcil de mantenir).

5.- Guia d'estil

Una guia d'estil de codi és un document que descriu un conjunt de regles i recomanacions a seguir quan escrivim codi en un llenguatge determinat. El PEP8 (https://www.python.org/dev/peps/pep-0008/) és l'especificació que recull la guia d'estil de Python. Si us plau, **llegiu ara aquest document** i intenteu seguir les indicacions que s'hi donen quan programeu en Python.

La guia s'ha d'interpretar com un conjunt de recomanacions, que cal seguir sempre i quan el sentit comú no indiqui el contrari. És a dir, no són normes estrictes a complir, i en certes circumstàncies serà preferible no fer cas a algunes de les recomanacions. En general, però, seguir les recomanacions de la guia farà que el codi que escrivim sigui fàcilment llegible, tant per nosaltres mateixos com per a altres desenvolupadors.

Existeixen diverses eines que ajuden als desenvolupadors a seguir les guies estil. En els notebooks de l'assignatura, farem servir `pycodestyle`, una eina que podem activar en els notebooks per tal de mostrar missatges d'alerta quan el codi que escrivim se salti les recomanacions de la guia d'estil. En primer lloc, carregarem l'extensió `pycodestyle_magic`. Després, l'activarem o desactivarem fent servir les instruccions `%pycodestyle_on` i `%pycodestyle_off`:

In [59]:

```
%load_ext pycodestyle_magic
```

In [60]:

```
%pycodestyle_on
```

In [61]:

```
# Codi que segueix la guia d'estil: no genera alertes
print("Wrong style")
```

```
Wrong style
```

In [62]:

```
# Codi que no segueix la guia d'estil: genera un missatge d'alerta
print("Wrong style" )
```

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

```
Wrong style
```

Fixeu-vos que les dues instruccions relatives a la revisió de l'estil de codi comencen pel caràcter `%`. Aquestes instruccions no són sentències de Python, sinó instruccions especials del kernel que es fa servir en els notebooks per proveir-los de funcionalitat addicional. Aquestes instruccions es coneixen en anglès com a *magic commands*.

6.- Exercicis per practicar

A continuació hi trobareu un conjunt de problemes que us poden servir per a practicar els conceptes explicats en aquesta primera unitat, així com per a refrescar els conceptes bàsics de programació. Us recomanem que intenteu fer aquests problemes vosaltres mateixos i que, una vegada realitzats, compareu la solució que us proposem amb la vostra solució. No dubteu en adreçar tots els dubtes que sorgeixin de la resolució d'aquests exercicis o bé de les solucions proposades al fòrum de l'aula.

1. Calcula la suma dels primers 100 números parells.

In [63]:

```
# Resposta
```

2. La seqüència de Fibonnacci és una seqüència en la qual cada terme és la suma dels dos termes anteriors (la seqüència comença amb els valors 1 i 1).

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

Calculeu la suma dels 50 primers valors de la seqüència.

In [64]:

```
# Resposta
```

3. Diem que un nombre és primer si només és divisible per ell mateix i per 1. Genereu una llista amb els 100 primers nombres primers.

In [65]:

```
# Resposta
```

4. Calculeu la suma dels quadrats dels números naturals entre 100 i 200 (ambdós valors inclosos).

In [66]:

```
# Resposta
```

5. Genereu tots els números de 3 dígits que es poden generar amb els dígits 3, 5 i 7.

In [67]:

```
# Resposta
```

6. Compteu i deseu el número de vegades que apareix cada paraula en el següent text.

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 [68]:

```
# Resposta
```

7. Genereu totes les possibles subcadenes de 3 lletres de la següent paraula: 'Electrodinamòmetre'.

In [69]:

```
# Resposta
```

8. Creeu una funció que retorni totes les possibles subcadenes d'n lletres d'una paraula qualsevol. La funció rebrà com a paràmetres el número de lletres `n` i la paraula `word`, i retornarà una llista amb les subcadenes.

In [70]:

```
# Resposta

```

6.1.- Solucions als exercicis per practicar

1. Calcula la suma dels primers 100 números parells.

In [71]:

```
# Opció 1: fent servir una list comprehension que filtra els
# valors parells (i%2 == 0) i sumant els elements de la llista amb sum:
suma = sum([i for i in range(200) if i % 2 == 0])
print(suma)
```

```
9900
```

In [72]:

```
# Opció 2: creant un bucle for amb un if a dins, de manera que només sumem
# els valors parells (i%2 == 0)
suma = 0
for i in range(200):
    if i % 2 == 0:
        suma = suma + i
print(suma)
```

```
9900
```

2. La seqüència de Fibonnacci és una seqüència en la qual cada terme és la suma dels dos termes anteriors (la seqüència comença amb els valors 1 i 1).

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

Calculeu la suma dels 50 primers valors de la seqüència.

In [73]:

```
# Opció 1: creant una llista amb els valors a sumar

# Creem una llista amb els 50 primers valors
fib = [1, 1]
for i in range(48):
    fib.append(fib[-1] + fib[-2])
print(fib)

# Sumem els valors de la seqüència
suma = sum(fib)
print(suma)
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597,
2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393, 196418, 3
17811, 514229, 832040, 1346269, 2178309, 3524578, 5702887, 9227465, 14
930352, 24157817, 39088169, 63245986, 102334155, 165580141, 267914296,
433494437, 701408733, 1134903170, 1836311903, 2971215073, 4807526976,
7778742049, 12586269025]
32951280098
```

In [74]:

```
# Opció 2: sense crear una llista amb els valors de la seqüència

suma = 2
# Desem els dos últims valors de la seqüència a les variables l1 i l2
l1, l2 = 1, 1
for i in range(48):
    n = l1 + l2
    suma += n
    l2 = l1
    l1 = n

print(suma)
```

```
32951280098
```

3. Diem que un nombre és primer si només és divisible per ell mateix i per 1. Genereu una llista amb els 100 primers nombres primers.

In [75]:

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

    # Si i és divisible per qualsevol dels primers que ja hem identificat,
    # aleshores i no és primer
    for p in primes:
        if i % p == 0:
            is_prime = False
            break
    # Si i és primer, l'afegim a la llista de primers
    if is_prime:
        primes.append(i)
    i += 2

print(primes)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 6
7, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 13
9, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 22
3, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 29
3, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 38
3, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 46
3, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541]
```

4. Calculeu la suma dels quadrats dels números naturals entre 100 i 200 (ambdós valors inclosos).

In [76]:

```
# Fem servir sum sobre una list comprehension que ens genera els valors
# dels quadrats demanats
sum([x**2 for x in range(100, 201)])
```

Out[76]:

```
2358350
```

5. Genereu tots els números de 3 dígits que es poden generar amb els dígits 3, 5 i 7.

In [77]:

```
digits = [3, 5, 7]
# Creem tres bucles for que recorren la llista de possibles dígits
result = []
for d1 in digits:
    for d2 in digits:
        for d3 in digits:
            result.append(100*d1 + 10*d2 + d3)

print(result)
```

```
[333, 335, 337, 353, 355, 357, 373, 375, 377, 533, 535, 537, 553, 555,
557, 573, 575, 577, 733, 735, 737, 753, 755, 757, 773, 775, 777]
```

6. Compteu i deseu el número de vegades que apareix cada paraula en el següent text.

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 [78]:

```
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)"

# Fem servir un diccionari per emmagatzemar el número de vegades que
# surt cada paraula
word_counter = {}
for word in text.split(" "):
    if word in word_counter:
        # Si ja hem trobat la paraula anteriorment, incrementem el comptador
        word_counter[word] += 1
    else:
        # Si és la primera vegada que vegem la paraula, posem el comptador a 1
        word_counter[word] = 1

print(word_counter)
```

```
{'Forty-two': 1, 'is': 1, 'a': 1, 'pronic': 1, 'number': 2, 'and': 2,
'an': 1, 'abundant': 1, 'number;': 1, 'its': 1, 'prime': 1, 'factoriza
tion': 1, '2': 1, '·': 4, '3': 2, '7': 1, 'makes': 1, 'it': 1, 'the':
3, 'second': 2, 'sphenic': 1, 'also': 1, 'of': 1, 'form': 1, '(2': 1,
'r)': 1}
```

7. Genereu totes les possibles subcadenes de 3 lletres de la següent paraula: 'Electrodinamòmetre'.

In [79]:

```
word = "Electrodinamòmetre"
words = []
# Extraiem la subcadena de 3 lletres que comença en cada possible
# posició inicial
for i in range(len(word)-2):
    words.append(word[i:i+3])
print(words)
```

```
['Ele', 'lec', 'ect', 'ctr', 'tro', 'rod', 'odi', 'din', 'ina', 'nam',
'amò', 'mòm', 'òme', 'met', 'etr', 'tre']
```

8. Creeu una funció que retorni totes les possibles subcadenes d'n lletres d'una paraula qualsevol. La funció rebrà com a paràmetres el número de lletres `n` i la paraula `word`, i retornarà una llista amb les subcadenes.

In [80]:

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

In [81]:

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

Out[81]:

```
['Elec',
 'lect',
 'ectr',
 'ctro',
 'trod',
 'rodi',
 'odin',
 'dina',
 'inam',
 'namò',
 'amòm',
 'mòme',
 'òmet',
 'metr',
 'etre']
```