القوائم (Lists)

تُعَدُّ القوائم إحدى المزايا المُهِمَّة للغات البرمجة؛ إذ تُسهِم في إدارة تنظيم البيانات، وتتيح للمُستخدِم التعامل معها واسترجاعها بسهولة.

حيث تمتلك بايثون عددًا من أنواع البيانات المركبة والتي تستخدم لتجميع القيم الأخرى مع بعضها البعض، و**القوائم** هي أوسع هذه الأنواع وأكثرها شمولًا. وكما تعلمنا سابقًا، يمكن كتابتها كقائمة من القيم (العناصر) المفصولة عن بعضها البعض بفواصل[، ومحاطة بأقواس مربعة **[،]**].

تُمثِّل **القائمة** في لغة البرمجة بايثون (**Python**) عددًا من القِيَم التي يُخزَّن بعضها مع بعض، وترتبط معًا بمعنى وظيفي مشترك. فمثلًا، يُمكِن للمُستخدِم تخزين أسماء الشوارع داخل إحدى المدن في قائمة، وكذلك تخزين أسعار البضائع التي اشتراها أحد العملاء، أو تخزين علامات الطلبة في الصف الحادي عشر.

تعريف القائمة (Python) يتطلَّب استخدام القوائم في لغة البرمجة بايثون أوَّلًا، ثمَّ تخزين العناصر داخلها.

يُمكِن تعريف إحدى القوائم باستخدام **الأقواس المُربَّعة** **[ ]**، والفصل بين عناصر القائمة **بفواصل** على النحو الآتي:

```
mylist = ['value1','value1', 'value1','value1']
```

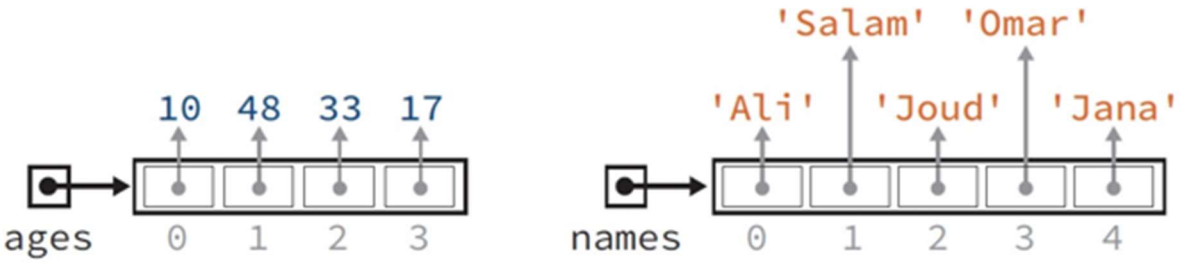
إسم القائمة **mylist** حيث تمثل
value1 و تمثل قيمة ثابتة تتكرر

مثال:

يُبيِّن المقطع البرمجي الآتي تعريفًا لقائمة تُسمّى **(names)**، وتتضمَّن **5** أسماء، وتعريفًا لقائمة أُخرى تُسمّى **(ages)**، وتتضمَّن **(4)** أعمار لأشخاص وتعريفا لقائمة فارغة كما يلي:

```
names = ['Ali', 'Salam', 'Joud', 'Omar', 'Jana']
ages = [10, 48, 33, 17]
a = []
```

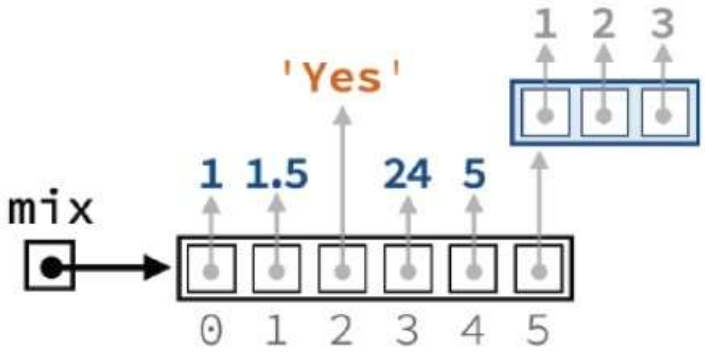
ويظهر الشكل التالي ترتيب العناصر في ذاكرة الكمبيوتر التي تم تعريفهاداخل القائمتين



ملاحظة:

لا يُشترَط في القائمة التي يراد إنشاؤها أنْ تحوي جميعها عناصر من النوع نفسه؛ إذ يُمكِن للمُستخدِم تخزين أرقام صحيحة، وأرقام عشرية، وسلاسل نصية، وقوائم أُخرى في القائمة نفسها كما في **المثال الآتي**:

```
[[ 1, 2, 3 ] , mix = [1, 1.5, 'Yes', 24, 5
```



أُلاحِظ من المثال السابق أنَّ طول القائمة هو (6) عناصر، بالرغم من أنَّ بعض هذه العناصر مُركَّبة من عناصر أخرى

ملاحظة:

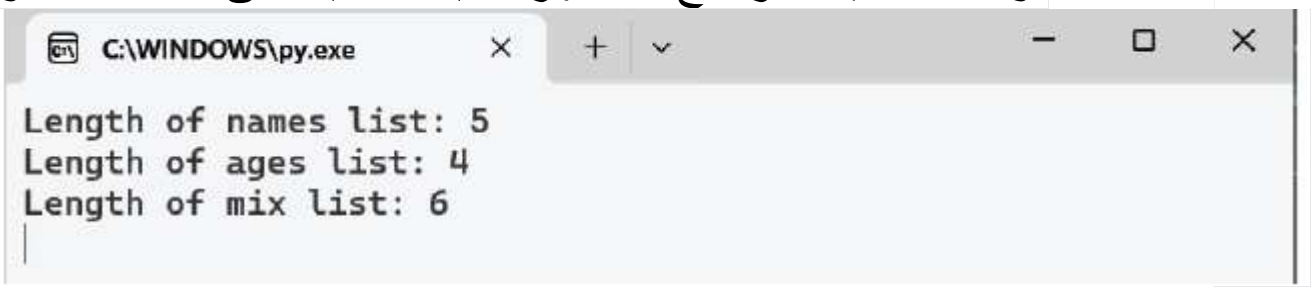
كما في المثال **()len** يُمكِن للمُستخدِم معرفة طول أيِّ قائمة باستخدام **الدالة** التالي:

```
len length_of_names = len(names)
length_of_ages = len(ages)
length_of_mix = len(mix)
```

print () : و لطباعة طول كل قائمة نستخدم **الدّالة**

```
print("Length of names list:", length_of_names)
print("Length of ages list:", length_of_ages)
print("Length of mix list:", length_of_mix)
```

و عند تشغيل البرنامج، ستظهر النتيجة الآتية على شاشة الحاسوب:



نشاط عملي

أُجرِّب وأستكشف:

أعمل – بالتعاون مع أفراد مجموعتي– على اســـتخدام لغة البرمجة بايثون (Python)، لإنشاء قائمة تضمُّ أســـماء طلبة المجموعة، وقائمة أُخرى تُبيِّن هواياتهـــم المُفضَّلة. بعد ذلك أكتب الأوامر البرمجية، وأتحقَّق من صحتها عبر طباعة طول كل من القوائم وطباعة أســـماء الطلاب مع هواياتهم المفضلة.

الحل:

```
[1]  اسماء_الطلبة = ["أحمد", "محمد", "خالد", "سارة", "نورة"]
     هوايات_الطلبة = ["القراءة", "السباحة", "كرة القدم", "الرسم", "الطبخ"]
     print("عدد الطلبة:", len(اسماء_الطلبة))
     print("عدد الهوايات:", len(هوايات_الطلبة))
     for i in range(len(اسماء_الطلبة)):
       print(اسماء_الطلبة[i], ": ", هوايات_الطلبة[i])
```

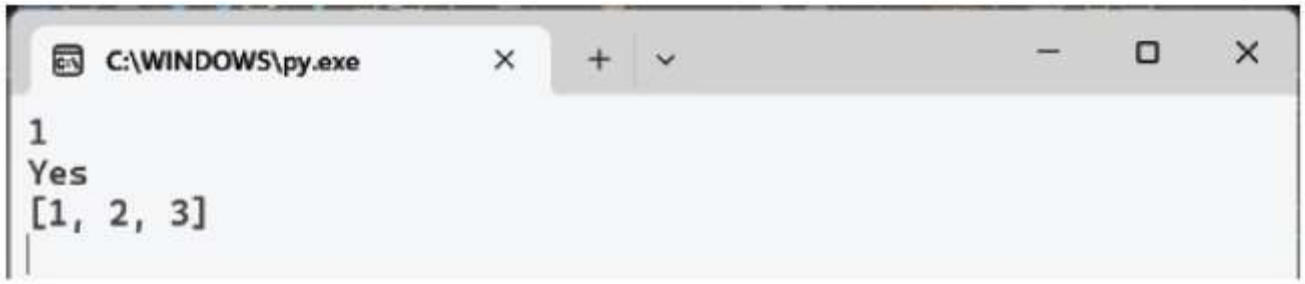
```
عدد الطلبة: 5
عدد الهوايات: 5
أحمد : القراءة
محمد : السباحة
خالد : كرة القدم
سارة : الرسم
نورة : الطبخ
```

الوصول للعناصر في القائمة:

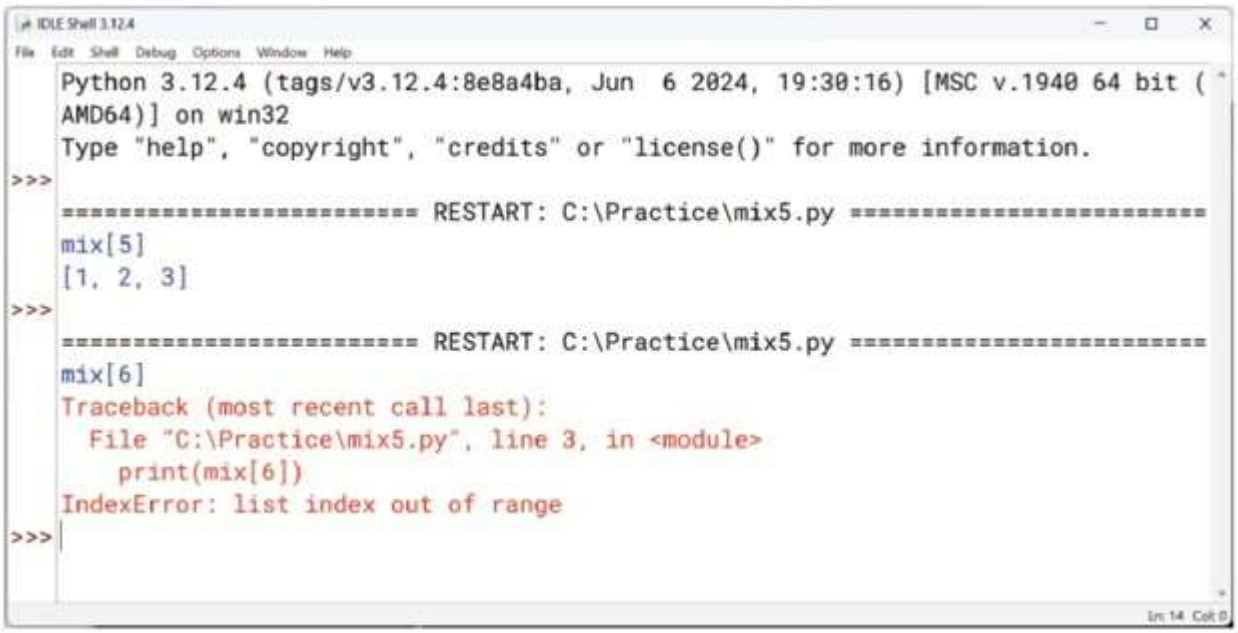
يُمكِن للمُستخدِم الوصول إلى أيِّ عنصر من عناصر القائمة باستخدام الأقواس المُربَّعة ورقم يُمثِّل موقع العنصر (**index**) في القائمة على النحو الآتي :

```
mix = [1, 1.5, 'Yes', 24, 5, [1, 2, 3]]
print(mix[0])
print(mix[2])
print(mix[5])
```

عند تشغيل البرنامج، ستظهر النتيجة الآتية على شاشة الحاسوب:



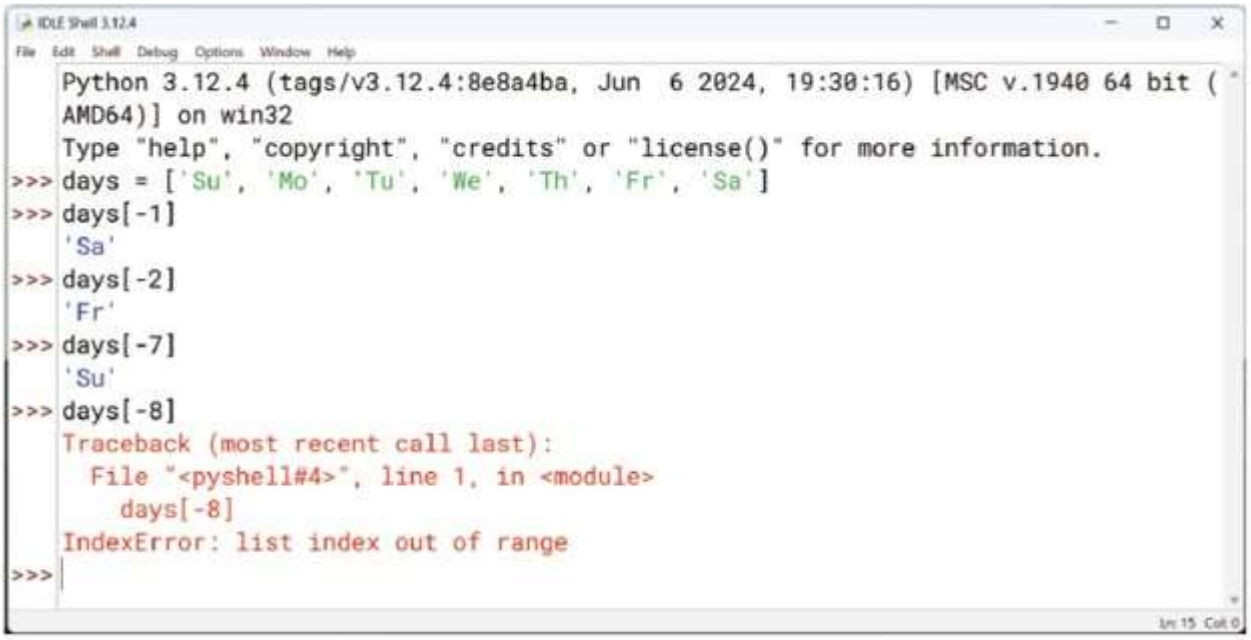
وهذا (1). أُلاحِظ أنَّ ترقيم مواقع العناصر قد **بدأ بالرقم (0)، لا بالرقم (1)** سيكون أخر عنصر فيها في (**mix**) يعني أنَّ قائمة من ستة عناصر مثل (**6**) لا في الموقع (**5**)الموقع من(5)، فإنَّ ذلك سيؤدّي إلى توقُّف البرنامج عن العمل، كما يظهر معنا في الشكل التالي:



للمُستخدِم استعمال أرقام سالبة (Python) كذلك تتيح لغة البرمجة بايثون بحيث **تشير الأرقام السالبة إلى العد العكسي** للوصول إلى العناصر، **إذ** تبدأمن نهاية القائمة:

مثال

الرقم(1-) يشير إلى العنصر الأخير، ويرمز الرقم (2-) إلى العنصر قبل الأخير وهكذا، حسب ما يظهر في الشكل:



نشاط:

نشاط عملي

- أكتب المقطع البرمجي اللازم لتعريف قائمة (months)، ثمَّ أُضيف العناصر إليها.
- أستخدِم موقع العنصر للوصول إلى العنصر ('May').
- أستخدِم الأعداد السالبة للوصول إلى العنصر ('Jun').
- أُنفِّذ المقطع البرمجي للتحقُّق من صحته، وأتتبَّع الأخطاء، وأعمل على تصحيحها.

الحل:

```
[2] months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"]

    # الوصول إلى العنصر "May" (العدد 4) باستخدام موقعه
    print(months[4])

    # الوصول إلى العنصر "Jun" (العدد -6) باستخدام الأعداد السالبة
    print(months[-6])
```

```
May
Jul
```

إضاءةٌ

استخدام الرقم (1−) يُقلِّل من نسبة حدوث الخطأ مقارنةً باستخدام موقع العنصر الأخير، مثل الرقم (6) في القائمة (days)؛ فالرقم (1−) يشير دائمًا إلى العنصر الأخير بِغَضِّ النظر عن طول القائمة، ويُسهِّل على الجميع فهم العنصر المقصود خلافًا للرقم (6) مثلًا؛ فهذا الرقم يتطلَّب من قارئ البرنامج معرفة أنَّ طول القائمة هو (7)، واستنتاج أنَّ الرقم (6) يُمثِّل موقع العنصر الأخير.

المرور على القوائم:

بما أن العناصر داخل القوائم تكون مرتبة فيمكن الوصول لعنصر معين قد يَلزم في بعض البرامج داخلها عن طريق ترتيبه داخل القائمة،ولكن أحيانًا المرور على جميع العناصر في القائمة لتفيذ وظيفة معينة كما يحتاج البرنامج،

على سبيل المثال:

البرنامج التالي هدفه طباعة كل عنصر من عناصر القائمة المُسمّاة(items) على سطر مُنفصِل:

```
for e in items:
    print(e)
```

كذلك يُمكِن كتابة البرنامج السابق باستخدام نوع آخر من حلقات التكرار، هو حلقة(while) كما يلي:

```
length = len(items)
i = 0
while i < length:
    print(items[i])
    i += 1
```

يُلاحَظ على هذا البرنامج ما يأتي:

1. استعمال المُتغيِّر **(i)** للمرور على جميع مواقع العناصر في القائمة؛ إذ بدأت قيمة هذا المُتغيِّر عند الرقم (0)، وهو موقع العنصر الأوَّل، وانتهت قيمته عند آخر موقع عنصر في القائمة.
2. الحاجة لمعرفة عدد المواقع التي سنقوم بالمرور عليها ؛ ما ألزم استخدام **len(items)** التي تُبيِّن طول القائمة المُخزَّنة في (items)
3. استخدام شرط حلقة التكرار يتأكد من عدم وصول قيمة المتغير **(i)** لطول القائمة، وذلك لأنه لو كان طول القائمة 10 على سبيل

المثال فإن آخر عنصر في القائمة موجود في الموقع 9 وليس في الموقع 10 .

مثال:

في الشكل التالي تتضمَّن قراءات (**readings**) إذا افترضْتُ أنَّ القائمة لجهاز استشعار، وأنَّ القراءات المتتابعة والمتساوية تدلُّ على وجود خطأ ما في القراءة، فإنَّه يُمكِنني التحقُّق من وجود عناصر متتابعة ومتساوية في القائمة كما يأتي:

```
found = False
for i in range(len(readings) - 1):
    if readings[i] == readings[i+1]:
        found = True
        break

print(found)
```

نشاط

أُحلِّل وأستنتج: استنادًا إلى المثال الوارد في الشكل (5-6):

- ما سبب استخدام جملة التحكُّم (break) في السطر (5)؟ وما تأثير حذفها في ناتج تنفيذ البرنامج؟
- ما دلالة استخدام الجملة range(len(readings)-1)؟ وما تأثير تعديلها إلى range(len(readings)) في ناتج تنفيذ البرنامج؟

أُناقِش زملائي/ زميلاتي ومُعلِّمي/ مُعلِّمتي في إجاباتي.

الحل :

1. يتم انهاء الحلقة في حال تحقق الشرط : **readings[i] == readings[i+1]**
2. الأمر **range(len(readings) - 1)** : يُستخدم لتحديد نطاق التكرار في الحلقة، بحيث نقارن كل عنصر في القائمة بالعنصر الذي يليه، دون محاولة الوصول إلى عنصر غير موجود.

إضاءة

الخطأ المعروف باسم "off-by-one error"، أو باسم الخطأ بخطوة واحدة، هو من أكثر الأخطاء البرمجية شيوعًا؛ قد يُخطِئ المُبرمِج عند تحديد شرط نهاية الدوران في حلقة التكرار (while)، أو عند تحديد المدى في حلقة التكرار (for)؛ ما يدفع البرنامج إلى تنفيذ دورة واحدة أقل من المطلوب أو أكثر منه.

إضاءة

يوجد نمط برمجي يســـتخدم متغير يلعب دور الراية (flag)، ويعتمد في كتابة البرامج على ما يأتي:

1- البَدْء بافتراض نتيجة بحث.

2- المرور على القائمة للتحقُّق من صحة الفرضية.

3- طباعة النتيجة بعد الانتهاء من المرور على القائمة.

في المثال السابق، أُطلِق على المُتغيِّر (found) اسم الراية (flag) التي تُرفَع أو تُنزَل عند اكتشاف وجود صفة ما أثناء عملية المرور. وهذا النمط في كتابة البرامج مفيد في عملية البحث، وله استخدامات كثيرة.

نشاط

أُناقِش وأُجرِّب:

كيف يُمكِن المرور على عناصر قائمة بالعكس؟ أُناقِش زملائي/ زميلاتي ومُعلِّمي/ مُعلِّمتي في هذا السؤال، ثمَّ أُجرِّب تنفيذ ذلك عمليًّا في بيئة بايثون (Python).

الحل:

العمليات في القوائم:

توجد عمليات عِدَّة يُمكِن تنفيذها في القوائم، مثل: الوصول إلى العناصر في قائمة ما، وإضافة عناصر جديدة إليها، وحذف عناصر منها، وترتيب العناصر فيها. وقد نُفِّذت بعض هذه العمليات بصورة فعلية في الأمثلة السابقة لهذا الدرس

نشاط

أُجرِّب وأُناقِش: أُجرِّب تنفيذ كل جملة من الجمل الآتية:

```
[1, 2, 3] + [98, 99, 100]
[1, 2, 3] + 5
[1, 2, 3] + [5]
[1, 2, 3] * 4
[1, 2, 3] * [1, 2, 3]
```

```
[13] list1 = [1, 2, 3]
     number = 5
     list2 = list1 + [number]
     print(list2)
```

```
[1, 2, 3, 5]
```

```
[3]  list1 = [1, 2, 3]
     list2 = [98, 99, 100]
     list3 = list1 + list2
     print(list3)
```

```
[1, 2, 3, 98, 99, 100]
```

```
[1]  my_list = [1, 2, 3, 4, 5]
```

```
[16] my_list = [1, 2, 3]
     result = my_list * 4
     print(result)
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
list1 = [1, 2, 3]
list2 = [5]
list3 = list1 + list2
print(list3)
```

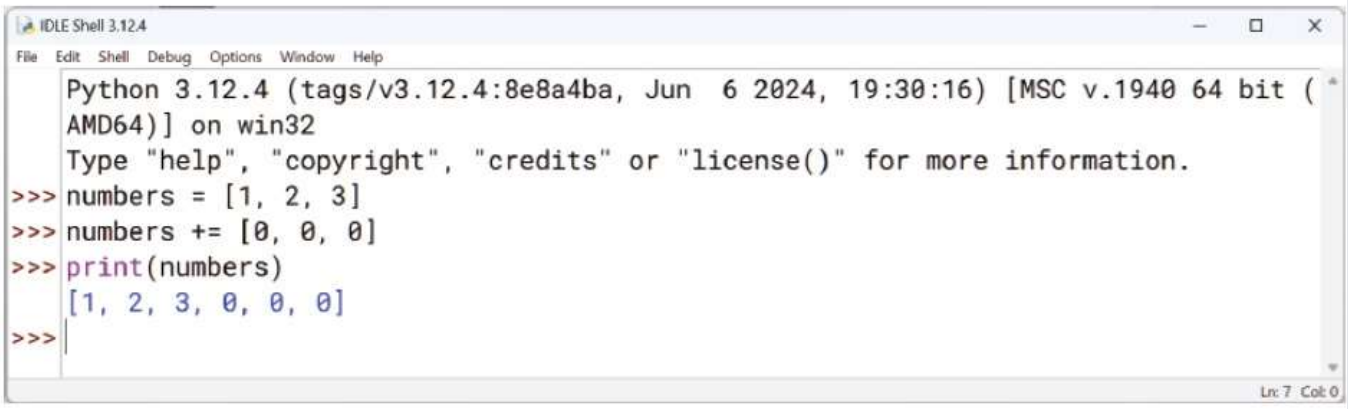
```
[1, 2, 3, 5]
```

```
[21]  list1 = [1, 2, 3]
      list2 = [1, 2, 3]
      result = []
      for x, y in zip(list1, list2):
          result.append(x * y)
      print(result)
```

```
[1, 4, 9]
```

```
[15]  [1, 2, 3] * 4
```

بين قائمتين، تنتج قائمة جديدة تحوي عناصر القائمتين معًا. ومن ثَمَّ يُمكِن استعمال العامل (+) عند استخدام عامل الجمع (=+) لتعديل قائمة، بإضافة عناصر قائمة أُخرى إلى نهايتها كما يأتي:



كذلك يُمكِن استخدام هذا العامل في إنشاء قائمة خطوة بخطوة كما في الشكل التالي:؛ حيث يقوم البرنامج الموضح في المثال بقراءة أرقام وإضافتها إلى قائمة ويتوقف عند إدخال أي رقم سالب.

```
numbers = []
n = int(input("Enter a positive number: "))
while n >= 0:
    numbers += [n]
    n = int(input("Enter a positive number: "))
print(numbers)
```

:أُلاحِظ

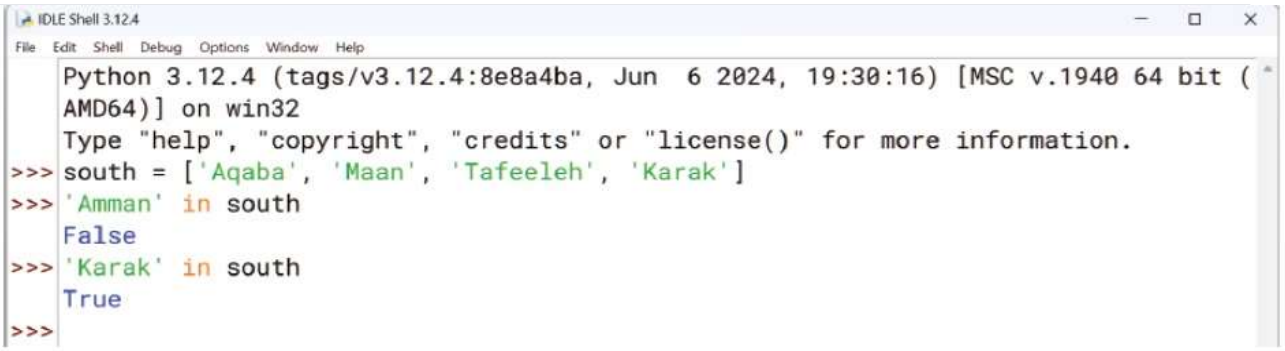
- هذا البرنامج بدأ بقائمة فارغة، ثمَّ أُضيفت العناصر تِباعًا باستخدام العامل (=+).
- يؤدّي **(=+)**؛ لأنَّ العامل **[n]**أُحيطت به الأقواس المُربَّعة (n) أنَّ الرقم (من **numbers += n** وظيفته بين قائمتين؛ مما يجعل الجمل، مثل جملة دون أقواس)، غير صحيحة بحسب قوانين لغة بايثون.

- (الذي يعمل على تكرار قائمة ما عددًا من المَرّات)، يُمكِن أيضًا المقارنة بين القوائم باستخدام عوامل المقارنة المنطقية**(== و =! و >و <= و <و =>و)**، التي تعمل على مقارنة كل عنصر في القائمة الأولى بالعنصر الذي يُقابِله في القائمة الثانية. والعامل ***** إضافةً إلى العامل **+**

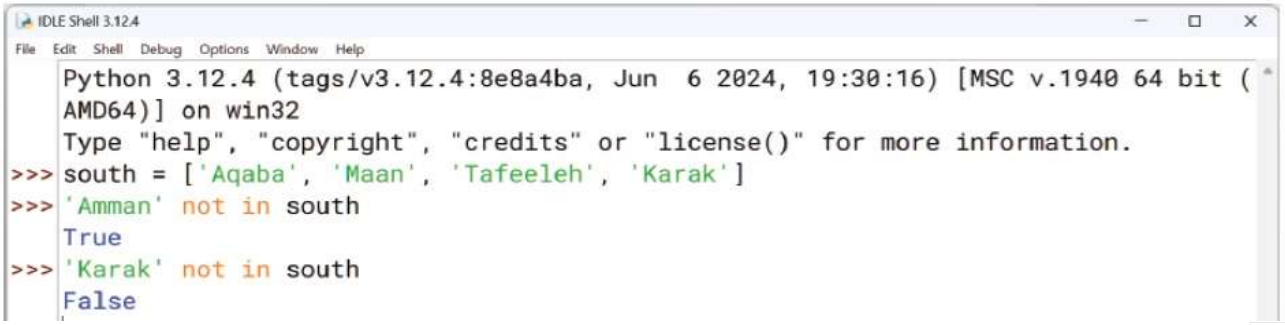
:مثال

- من القائمة [1 , 3]؛ لأنَّ العنصر الأوَّل(1) في [1 , 2 , 3] < القائمة القائمة الأولى أقل من العنصر الأوَّل في القائمة الثانية(3).

- من القائمة [0 , 0 , 0 , 1]؛ لأنَّ [1 , 2 , 3] < في حين أنَّ القائمة في (0)العنصر الثاني (2) في القائمة الأولى أكبر من العنصر الثاني في القائمة الثانية.
- كذلك يُمثِّل التحقُّق من وجود عنصر ما في القائمة واحدةً من العمليات المُهِمَّة على النحو (in)في القوائم. وأسهل طريقة لعمل ذلك هي استخدام العامل الآتي:



يُمكِن أيضًا استخدام العامل (not in) في التحقُّق من عدم وجود عنصر ما في القائمة كما يأتي:



نشاط عملي

أُجرِّب وأستكشف:

أعرّف قائمتين فارغتين ثم أضيف خمسة أعداد إلى كل منهما.

أستخدم عوامل المقارنة لطباعة عناصر القائمة الأصغر.

أستخدِم معامل الجمع في ضَمِّ عناصر القائمتين وطباعتهما.

أُنفِّذ البرنامج، ثمَّ أتحقَّق من صحته، وأعمل على استكشاف الأخطاء وتصحيحها.

الحل:

أعرّف قائمتين فارغتين ثم أضيف خمسة أعداد إلى كل منهما:

```
L1 =[]
L2 = []
# إضافة 5 أعداد إلى القائمة1
for i in range(5):
    L1.append(int(input("أدخل عددًا للقائمة1: ")))
# إضافة 5 أعداد إلى القائمة2
for i in range(5):
    L2.append(int(input("أدخل عددًا للقائمة2: ")))
```

أستخدم عوامل المقارنة لطباعة عناصر القائمة الأصغر:

```
if len(L1) > len(القائمة2):
    print("عناصر القائمة الأصغر", L1 ,":(L1))
elif len(L1) < len(L2):
    print("عناصر القائمة الأصغر", L2 ,":(L2))
else:
    print("القائمتان متساويتان في الطول.")
```

أستخدِم معامل الجمع في ضَمِّ عناصر القائمتين وطباعتهما:

```
L3 = L1+L2
print("عناصر القائمة المجمعة:", L3)
```

الدوالُّ الجاهزة لمعالجة القوائم:

في تحديد طول قائمة ما، والآن len (...) تعرَّفْتُ سابقًا أنَّه يُمكِن استخدام سأتعرَّف أنَّ العديد من الدوالِّ المُشابِهة تُوفِّرها اللغة للتعامل مع القوائم، أنظر

الجدول (5- 1) الذي يعرض أمثلة على ذلك، علمًا بأنَّ جميع هذه الأمثلة مُطبَّقة على القائمة: [4 , 2 , 3 , 5 , 3] = numbers.

الجدول التالي يحتوي 8 عأمثلة على بعض الدوالِّ الجاهزة لمعالجة القوائم.

الشرح	المثال
تعمل max(...) على إيجـــاد أكبر عنصر في القائمة بافتراض أنَّ جميع العناصر هي من النوع نفسه.	`>>> max(numbers)` `5`
تعمل min(...) علـــى إيجاد أصغر عنصر في القائمة بافتراض أنَّ جميع العناصر هي من النوع نفسه.	`>>> min(numbers)` `1`

تعمل sorted(...) على إعادة نسخة مُرتَّبة من القائمة (القائمة الأصلية تظلُّ من دون تعديل).	`>>> result = sorted(numbers)`
	`>>> print(result)`
	`[1, 2, 3, 3, 4, 5]`
	`>>> print(numbers)`
	`[1, 3, 5, 3, 2, 4]`

تعمل count(...) على عَدِّ عدد مَرّات تكرار عنصر ما في القائمة.	`>>> numbers.count(3)`
	`2`
	`>>> numbers.count(7)`
	`0`

تعمل index(...) على إيجاد موقع أوَّل ظهور لعنصر ما في القائمة.	`>>> print(numbers)`
	`[1, 3, 5, 3, 2, 4]`
	`>>> numbers.index(5)`
	`2`
	`>>> numbers.index(3)`
	`1`

يتبيَّن من الجدول السابق وجود طريقتين مختلفتين لاستخدام الدوالِّ الجاهزة:

- الطريقة الأولى تُستخدَم في (min)، و(max)، و(sorted)، و(len)، وتتمثَّل في إرسال القائمة إلى الدالة(print).

- الطريقة الثانية تُستخدَم في (count)، و(index)، وتتمثَّل في استدعاء الوظيفة عن طريق ذكر اسم القائمة متبوعًا بنقطة، ثمَّ ذكر اسم الدالة.

يُذكَر أنَّ الدوال في كلتا الطريقتين لا تقوم بإجراء أيِّ تعديل على القائمة، وإنَّما تكتفي بالمرور على القائمة لحساب نتيجة ما، ثمَّ إعادة هذه النتيجة. غير أنَّ ذلك ليس مُطَّرِدًا في جميع الدوال، كما هو الحال في تلك الواردة في الجدول التالي؛ إذ تعمل جميعها على تعديل القائمة.

الجدول التالي يحتوي أمثلة على وظائف جاهزة لمعالجة القوائم وتعديلها.

الشرح	المثال

تعمل sort() على ترتيب عناصر القائمة (تُعدَّل القائمة نفسها بدلًا من إرجاع نسخة مُرتَّبة كما في sorted(...)).	`>>> numbers = [1, 3, 5, 3, 2, 4]`
	`>>> numbers.sort()`
	`>>> print(numbers)`
	`[1, 2, 3, 3, 4, 5]`

تعمل reverse() على عكس ترتيب العناصر في القائمة.	`>>> numbers = [1, 3, 5, 3, 2, 4]`
	`>>> numbers.reverse()`
	`>>> print(numbers)`
	`[4, 2, 3, 5, 3, 1]`

تعمل append(...) على إضافة عنصر في آخر القائمة.	`>>> numbers = [1, 3, 5, 3, 2, 4]`
	`>>> numbers.append(99)`
	`>>> print(numbers)`
	`[1, 3, 5, 3, 2, 4, 99]`

تعمل remove(...) على حذف أوَّل ظهور لعنصر في القائمة.	`>>> numbers = [1, 3, 5, 3, 2, 4]`
	`>>> numbers.remove(3)`
	`>>> print(numbers)`
	`[1, 5, 3, 2, 4]`
	`>>>`

إضاءةٌ

من المفيد جدًّا تعرُّف أهمِّ الدوال الجاهزة التي تُوفِّرها اللغة، لكنَّ معظم المُبرمِجين المحترفين لا يحفظون جميع هذه الدوال، وإنَّما يعودون مِرارًا إلـــى الموقع الإلكتروني للغة البرمجة بايثون (Python)؛ للبحث عن الدوال المناسبة لبرامجهم، أو للتحقُّق من كيفية استخدام بعض هذه الدوال.

أتعرَّف الوظائف الجاهزة التي لها تعلُّق بالقوائم عن طريق الرابط الإلكتروني الآتي:

https://docs.python.org/3/tutorial/datastructures.html

أو عن طريق مسح الرمز سريع الاستجابة (QR Code) المجاور.

نشاط عملي

أُجرِّب وأستكشف:

- أستخدِم الدوالَّ الجاهزة في طباعة أكبر عنصر وأصغر عنصر في القائمتين اللتين أنشأْتُهما في النشاط السابق.
- أستخدِم الدوالَّ الجاهزة في طباعة عناصر القائمة الأولى بشكل عكسي.

```
list1 = [1,6, 2, 3]
list2 = [1, 2, 3]
largest_number = max(list1)
largest_number = max(list2)
smallest_number = min( list1)
smallest_number = min( list2)
print( largest_number)
print( smallest_number)
list1.reverse()
print(list1)
```

```
3
1
[3, 2, 6, 1]
```

سلاسل الحروف (Strings):

يوجد تشابه بين سلاسل الحروف والقوائم، يتمثّل في أنَّ كلتيهما تتألَّف من عناصر متتابعة. غير أنَّ جميع العناصر في السلسلة هي حروف (أيْ رموز)؛ لذا يُمكِنني تطبيق معظم ما تعلَّمْتُه عن القوائم على سلاسل الحروف (تُمثِّل حروفًا أبجديةً، أو أرقامًا، أو علاماتِ ترقيم، أو غيرَ ذلك).

نشاط فردي

أُجرِّب وأستكشف:

أُعرِّف سلسلة الحروف `name = 'Amina-Abdu'`، ثمَّ أُجرِّب تنفيذ الجمل الآتية:

```
name[0]
sorted(name)
max(name)
name += 'L'
name * 2
name < 'Zaid'
'Abd' in name
name[5] = 'h'
name.sort()
```

- أيُّ هذه الجمل تمكَّنْتُ من تنفيذها؟
- أيُّ هذه الجمل لم أتمكَّن من تنفيذها؟
- ما العامل المشترك بين الجمل التي لم أتمكَّن من تنفيذها؟

الحل:

الجمل التي لم تنفذ:

name : لم يتم تعريف متغير name قبل استخدامه. نحتاج إلى تعيين قيمة لـ name قبل استخدامه. على سبيل المثال، "name = "Ahmed.

'h' = [5]name: هذه العملية غير مدعومة في السلاسل النصية لأنها غير قابلة للتغيير. نحتاج إلى استخدام طريقة أخرى، مثل استبدال name بسلسلة نصية جديدة.

()name.sort: هذه الطريقة تعمل فقط مع القوائم، وليس مع السلاسل النصية.

العامل المشترك بين الجمل التي لم أتمكَّن من تنفيذها:

تعريف المتغيرات قبل استخدامها.

معرفة نوع البيانات التي نتعامل معها.

استخدام الوظائف والطرق المناسبة لنوع البيانات

يُمكِن الوصول إلى أيِّ حرف في السلسلة باستخدام الموقع (index)، ويُمكِن أيضًا استخدام الدوالِّ، مثل: (sorted) و (max)، و (len)، كما تعلَّمْنا سابقًا، (+) و (*) وكذلك استخدام العوامل، مثل: (count) و(in) فضلًا عن إمكانية اختبار وجود عنصر في السلسلة باستخدام العامل (not in)، والعامل (in) بالرغم من وجود فروق بسيطة بينها؛ إذ تتحقَّق العوامل في القوائم من وجود عنصر ما في إحدى القوائم، في حين تتحقَّق العوامل في السلاسل من وجود سلسلة أُخرى داخل السلسلة نفسها.

مثال:

يتحقَّق البرنامج الآتي من وجود السلسلة (' gov.jo .') ضمن السلسة (url):

```
url = input if '.gov.jo' in url ("ما موقعك الإلكتروني؟")
print ('يبدو أنك أدخلت موقعاً حكومياً أردنياً') :
:else
print ('شكراً جزيلاً')
```

الدوالُّ الجاهزة الخاصة بسلاسل الحروف:

توجد دوالُّ جاهزة تختصُّ بسلاسل الحروف، أنظر الجدول (5- 3) الذي يعرض أمثلة على ذلك، علمًا بأنَّ جميع هذه الأمثلة مُطبَّقة على السلسلة ':

```
.text = 'Hello there
```

الجدول (5-3): أمثلة على بعض الدوالّ الجاهزة لسلاسل الحروف.

الشرح	المثال

تعمل upper() على إنشاء نسخة من السلسلة، تحوي الحروف نفسها، ولكنْ بعد تحويل الأحرف الصغيرة إلى أحرف كبيرة.	`>>> text = 'Hello there'`
	`>>> text.upper()`
	`'HELLO THERE'`

تعمل lower() على إنشاء نسخة من السلسلة، تحوي الحروف نفسها، ولكنْ بعد تحويل الأحرف الكبيرة إلى أحرف صغيرة.	`>>> text.lower()` `'hello there'`

تعمل replace(a, b) على إنشاء نسخة من السلسلة بعد استبدال كل ظهور لـa بـb.	`>>> text.replace('Hello', 'Hi')`
	`'Hi there'`
	`>>> text.replace('e', 'X')`
	`'HXllo thXrX'`

تعمل isalpha() على التأكُّد أنَّ جميع حروف السلسلة هي حروف أبجدية، ومن ثَمَّ كانت نتيجة الاستدعاء الأوَّل هي (False)؛ نظرًا إلى وجود فراغ بين الكلمتين في السلسة، خلافًا للاستدعاء الثاني؛ إذ كانت نتيجته (True)؛ لأنَّ السلسلة تحوي فقط حروفًا أبجدية.
تعمل isnumeric() على التأكُّد أنَّ السلسلة تحوي فقط أرقامًا، في حين تعمل islower() وisupper() على التأكُّد أنَّ السلسلة تحوي فقط حروفًا أبجدية صغيرةً أو حروفًا كبيرةً.
	`>>> text.isalpha()`
	`False`
	`>>> text = "HelloThere"`
	`>>> text.isalpha()`
	`True`
	`>>> text.isnumeric()`
	`False`
	`>>> text = "HELLO"`
	`>>> text.islower()`
	`False`
	`>>> text.isupper()`
	`True`
	`>>>`

إضاءة

توجد عمليات أُخرى تُوفِّرها اللغة للتعامل مع السلاسل، ويُمكِنني تعرُّفها عن طريق الموقع الإلكتروني الرسمي للغة البرمجة بايثون (Python):

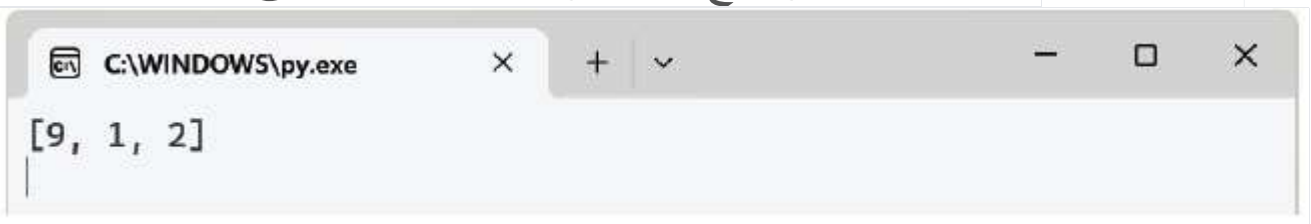
https://docs.python.org/3/library/stdtypes.html#string-methods

أو عن طريق مسح الرمز سريع الاستجابة (QR Code) المجاور.

تتمثَّل أهمُّ الفروق بين السلاسل والقوائم في أنَّ السلاسل في لغة البرمجة بايثون (Python) غير قابلة للتغيير (Immutable)، وأنَّ القوائم قابلة للتغيير (Mutable). ولهذا لا يُمكِن -مثلًا- تغيير أحد الحروف في السلسلة، في حين يُمكِن تغيير عنصر ما في القائمة، أنظر الشكل (5- 8).

```
numbers = [0, 1, 2]
numbers[0] = 9
print(numbers)
```

عند تشغيل البرنامج، ستظهر النتيجة الآتية على شاشة الحاسوب:



```
name = 'Jana'
name[0] = 'D'
print(name)
```

عند تشغيل البرنامج، ستظهر النتيجة الآتية على شاشة الحاسوب:

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    name[0] = 'D'
TypeError: 'str' object does not support item assignment
```

الشكل (5-8): مثال يُبيِّن الفرق بين السلاسل والقوائم.

أُلاحِظ أنَّ جميع العمليات في الشكل السابق لا تُعْنى بتعديل السلسلة، وإنَّما تُعْنى بإنشاء نسخة جديدة مُعدَّلة منها. فمثلًا، عند تنفيذ الجمل الآتية، فإنَّ السلسلة لن تتغيَّر:

```
>>> text = 'Hello there'
>>> text.replace('Hello', 'Hi')
'Hi there'
>>> text.upper()
'HELLO THERE'
>>> print(text)
Hello there
>>>
```

على السلسلة؛ إذ تُنشَأ upper () و replace () وهذا يتأكَّد عند استدعاء نسخ جديدة مُعدَّلة. ونظرًا إلى عدم حفظ هذه النسخ أو طباعتها؛ فإنَّ السلسلة لم تتأثَّر بهذه العمليات. وبالرغم من ذلك، يُمكِن حفظ النسخ المُعدَّلة للسلسلة كما في الجمل الآتية:

```
>>> text = 'Hello there'
>>> text = text.replace('Hello', 'Hi')
>>> text = text.upper()
>>> print(text)
HI THERE
>>>
```

كذلك يُمكِن المرور على عناصر القوائم المُركَّبة، وتطبيق بعض المهام عليها كما في القوائم البسيطة.

مثال:

عند إنشاء حساب جديد في موقع إلكتروني، فإنَّ هذا الموقع يتحقَّق من (جودة) كلمة السِّرّ؛ سعيًا لتقليل خطر اكتشافها من طرف المُخترِقين.

يتضمَّن هذا المثال إنشاء برنامج يعمل على استيفاء كلمة السِّرِّ للشرطين الآتيين:

- اشتمال كلمة السِّرِّ على (10) أحرف فأكثر.
- احتواء كلمة السِّرِّ على أحرف إنجليزية صغيرة، وأحرف إنجليزية كبيرة، وأرقام، ورموز غير الأحرف والأرقام.

يبدأ البرنامج المرور على أحرف كلمة السِّرِّ، وعَدَّ مَرّات تكرار كلِّ من الأحرف الأبجدية (الصغيرة والكبيرة) والأرقام والرموز، ثمَّ يتأكَّد أنَّ عدد مَرّات التكرار لكل ما سبق لا يساوي صفرًا، أنظر الشكل (9 -5).

```
psw = input("Enter Password: ")
small = 0
capital = 0
number = 0
special = 0

for c in psw:
    if c.islower():
        small += 1
    elif c.isupper():
        capital += 1
    elif c.isdigit():
        number += 1
    else:
        special += 1

if len(psw) < 10:
    print("Password must be >= 10 characters long.")

if small == 0:
    print("Password must contain small letters.")

if capital == 0:
    print("Password must contain capital letters.")

if number == 0:
    print("Password must contain numbers.")

if special == 0:
    print("Password must contain special characters.")

if small != 0 and capital != 0 and number != 0 and special != 0 and
len(psw) >= 10:
    print("Strong password!")
```

الشكل (5-9): مثال على سلسلة كلمة المرور.

نشاط عملي

أُنفِّذ البرنامج في المثال السابق في بيئة بايثون (Python)، وأُلاحِظ الناتج، ثمَّ أستكشف الأخطاء التي قد تحدث أثناء تنفيذي للبرنامج، وأعمل على تصحيحها.

عدم استخدام دالة input() بشكل صحيح:

يجب أن تكون input() داخل دالة أو حلقة ليتم تنفيذها بشكل متكرر حتى يتم إدخال كلمة مرور قوية. حالياً، يتم تشغيل input() مرة واحدة فقط، وإذا كانت كلمة المرور ضعيفة، لا يُطلب من المستخدم إعادة المحاولة.

يجب استخدام input() داخل حلقة while تظل تعمل حتى تصبح كلمة المرور قوية.

ترتيب الشروط:

يتم عرض رسائل الخطأ حتى لو كانت كلمة المرور قوية.

يجب عرض رسالة "Strong password!" فقط بعد اجتياز جميع الشروط.

يمكنك استخدام عبارة else بعد سلسلة من عبارات if لضمان عرض الرسالة الصحيح

نشاط جماعي

أستكشف وأُناقِش:

أتتبَّع الأوامر البرمجية في المثال السابق، وأستكشف إمكانية كتابتها بطريقة أُخرى، ثمَّ أُناقِش زملائي/ زميلاتي في أهمية كل أمر، وتأثير حذفه في تنفيذ البرنامج بصورة صحيحة.

```
:while True
psw = input("Enter Password: ")
small = 0
capital = 0
number = 0
special = 0
:for c in psw
:()if c.islower
small += 1
:()elif c.isupper
capital += 1
:()elif c.isdigit
number += 1
:else
special += 1
:10 > if len(psw)
(".characters long 10 =< print("Password must be
:elif small == 0
print("Password must contain small letters.")
```

```
:elif capital == 0
print("Password must contain capital letters.")
:elif number == 0
print("Password must contain numbers.")
:elif special == 0
print("Password must contain special characters.")
:else
print("Strong password!")
break # إيقاف الحلقة عند إدخال كلمة مرور قوية
```

حلقة while True: سيستمر البرنامج في طلب كلمة مرور جديدة حتى يتم إدخال كلمة مرور قوية.

استخدام elif: تضمن عدم عرض رسائل الخطأ غير الضرورية.

break: يوقف الحلقة عندما تكون كلمة المرور قوية.

باستخدام هذا الحل، سيُطلب من المستخدم إدخال كلمة مرور جديدة حتى يتم إدخال كلمة مرور قوية

إضاءة

يُمكِن كتابة برنامج لتشفير الرسائل بناءً على فكرة قديمة جدًّا، تُنسَب إلى القيصر الروماني يوليوس، الذي يقال إنَّه وظَّف هذه الفكرة في مراسلاته.

تقوم الفكرة على تمثيل كل حرف من الحروف الأبجدية بحرف آخر يبعد عنه مسافة مُحدَّدة، مثل استبدال كل حرف من الحروف الأبجدية بالحرف الذي يبعد عنه (13) خطوة؛ إذ يُستبدَل حرف الـ'a' بحرف الـ'n'، وحرف الـ'w' بحرف الـ'j'، وهكذا كما هو مُوضَّح في الشكل الآتي:

بناءً على ذلك، فإنَّ تشفير كلمة 'abu' هو 'noh'، وتشفير كلمة 'bad' هو 'onq'.

أُلاحِظ أنَّه لحساب موقع الحرف البديل يجب إضافة (13) خطوة، لكنَّ ذلك قد يجعل الموقع أكبر من (25) كما هو الحال بالنسبة إلى الحرف (w)؛ إذ سيكون الناتج هو (35) إنْ أُضيف (13) إلى موقع الحرف (22)، في حين يجب استبدال الحرف (w) بالحرف (j) الموجود في الموقع (9). ومن ثَمَّ يُمكِن طرح (13) بدلًا من إضافة (13)؛ ما يفي بالغرض (22 – 13= 9).

وتأسيسًا على ذلك، فإنَّ البرنامج سيظهر على النحو المُبيَّن في الشكل الآتي:

```
chars = 'abcdefghijklmnopqrstuvwxyz'
result = ''

text = input("Enter the text to encrypt/decrypt: ")

for c in text:
    if c.isalpha():                 # هل الحرف أبجدي؟
        c = c.lower()               # حول الحرف الكبير لحرف صغير
        i = chars.index(c)          # احسب موقع الحرف الأصلي

        new_i = i+13                # احسب موقع الحرف الجديد
        if new_i >= len(chars):
            new_i = i - 13

        c = chars[new_i]            # استبدل الحرف

    result += c                     # أضف الحرف للرسالة المشفرة

print(result)
```

الأحرف الخاصة:

؛ (Python) تحمل بعض الأحرف معاني خاصة في لغة البرمجة بايثون

لذا يجب الانتباه عند استخدامها في السلاسل. فمثلًا، تُستعمَل علامة التنصيص ' ' وعلامة التنصيص " " لتحديد بداية سلسلة الحروف ونهايتها؛ ما يجعل استخدام هذه العلامة حرفًا داخل السلسلة مشكلةً. ولهذا تسمحلغة بالتفرقة بين علامة التنصيص التي هي جزء من (Python) بايثون السلسلة وعلامة التنصيص التي تُحدِّد بداية السلسلة ونهايتها، وذلك عن طريق استخدام الرمز \ قبل علامة التنصيص كما يأتي:

```
>>> print('What is the difference between \' \' and " "? ')
What is the difference between ' ' and " "?
>>>
```

أُلاحِظ أنَّ الرمز \ لم يُضَف قبل علامة التنصيص "؛ لأنَّ العلامة التي استُخدِمت في تحديد بداية السلسلة ونهايتها هي علامة '.
كذلك يُستخدَم الرمز \ قبل بعض الحروف لإعطائها معنًى خاصًّا، مثل: '\ n' التي تعني سطرًا جديدًا، و 't\' التي تعني مسافة مُطوَّلة:

```
>>> print('Name:\tJamilah\nAge:\t16 years old')
Name:   Jamilah
Age:    16 years old
>>>
```

ولكنْ، كيف يُستخدَم الرمز \ حرفًا داخل السلسلة إنْ كان يحمل معنًى خاصًّا؟
يُمكِن فعل ذلك عن طريق إضافة رمز \ آخر قبله كما يأتي:

```
>>> print('I can print \\')
I can print \
>>>
```

نشاط فردي

أكتب أوامر برمجية، أستخدِم فيها الأحرف الخاصة، وأُنفِّذها، ثمَّ أُلاحِظ ناتج التنفيذ كل مَرَّة.

```
[2]  print("مرحباً \nبالعالم!")
     print("الاسم:\tمحمد")
     print("المسار: C:\\Users\\محمد")
     print("قال: \'السلام عليكم\'")
     print("قال: \"السلام عليكم\"")
     print(r"C:\Users\محمد")
```

```
مرحباً
بالعالم!
الاسم:	محمد
المسار: C:\Users\محمد
قال: 'السلام عليكم'
قال: "السلام عليكم"
C:\Users\محمد
```

القوائم المُركَّبة:

تعرَّفْتُ سابقًا أنَّ عناصر القائمة قد تحوي جملة من القوائم؛ ما يعني إمكانية إنشاء قائمة من مجموعة قوائم. وهذا النوع من القوائم المُركَّبة مُنتشِر ومُهِمٌّ لكثير من التطبيقات؛ إذ يُمكِن - مثلًا - في الرياضيات تمثيل المصفوفة في صورة قائمة مُركَّبة تحوي أرقامًا، وكذلك (2D Matrix)ثنائية الأبعاد تمثيل رقعة الشطرنج في صورة مصفوفة ثنائية الأبعاد تحوي أحجارًا، وغير ذلك كثير.

إنشاء القوائم المُركَّبة:

يُمكِن إنشاء قائمة مُركَّبة كما في الجملة الآتية:

```
a = [[0, 0, 0],
[0, 0, 0],
[0, 0, 0]]
```

أُلاحِظُ أنَّ هذه القائمة تتألَّف من (3) قوائم، وأنَّ كل قائمة منها تتألَّف من(3) عناصر؛ أيْ إنَّ هذه القائمة المُركَّبة تحوي (3) صفوف و(3) أعمدة.

، والوصول إلى الصف a[0] يُمكِن الوصول إلى الصف الأوَّل باستخدام وهكذا. كذلك يُمكِن الوصول إلى أيِّ عنصر من a[1،] الثاني باستخدام عناصر القائمة المُركَّبة عن طريق تحديد موقع الصف، ثمَّ تحديد موقع العنصر داخل هذا الصف)أيْ رقم العمود(. فمثلًا، العنصر الأوَّل في ، والعنصر الأخير في هذه a[0][0] القائمة المُركَّبة موجود في المكان القائمة موجود في المكان a[2][2] ، وهكذا.

مثال:

يعمل البرنامج الآتي على تعيين قيمة (99) للعنصر [2][1]، ثمَّ طباعة عناصر القائمة:

```
>>> a[1][2] = 99
>>> print(a)
[[0, 0, 0], [0, 0, 99], [0, 0, 0]]
```

مثال :

يطبع البرنامج الآتي عناصر القائمة المُركَّبة، ويضع كل صف على سطر مُنفصِل:

```
>>> for row in a:
...     print(row)

[0, 0, 0]
[0, 0, 99]
[0, 0, 0]
>>>
```

إنَّ هذه الطريقة اليدوية في تعيين قِيَم العناصر تُستخدَم فقط في إنشاء القوائم المُركَّبة الصغيرة. أمَّا القوائم المُركَّبة الكبيرة (مثل مصفوفة تتألَّف من (1000) صف و(2000) عمود) فيتطلَّب إنشاؤها استخدام حلقة تكرار، بَدْءًا بإعداد قائمة فارغة، وانتهاءً بإضافة كل صف إلى القائمة داخل حلقة التكرار كما يأتي:

```
a = []
for i in range(1000):
row = [0]*2000
a.append(row)
```

المرور على القوائم المُركَّبة:

لا يختلف المرور على القائمة المُركَّبة عن المرور على أيِّ قائمة أُخرى، لكنَّنا نحتاج كثيرًا إلى استخدام حلقات تكرار مُركَّبة (حلقة للمرور على كل صف داخلها، وحلقة للمرور على كل عنصر في الصف)؛ لأنَّ عناصر

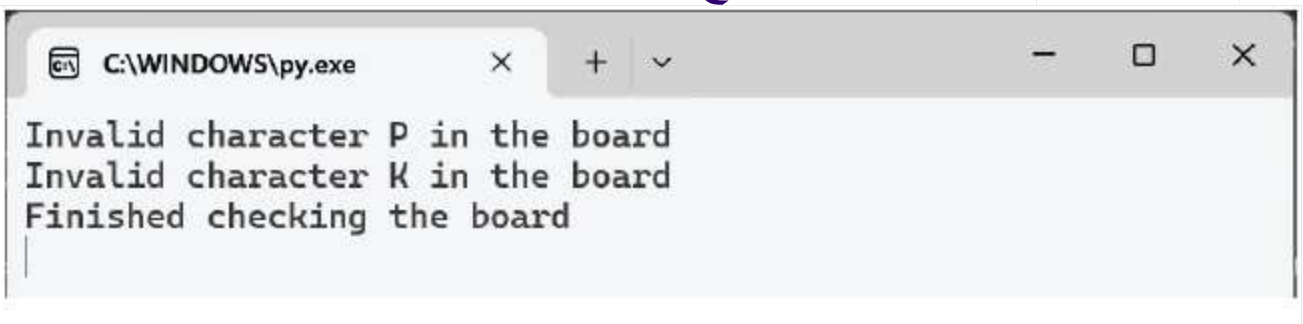
القائمة تتألَّف أساسًا من قوائم.

مثال:

يستخدم البرنامج الآتي حلقة التكرار في المرور على كل عنصر في رقعة (XO) مُخصَّصة للعبة (board اسمها)؛ بُغْيَةَ التحقُّق من عدم وجود أحرف في الرقعة، ما عدا 'X' أو 'O' أو '-':

```
board = [['-', 'X', 'P'],
['-', 'X', '-'],
['K', 'O', '-']]
for row in board:
    for c in row:
        if c != 'X' and c != 'O' and c != '-':
            print('Invalid character ' + c + ' in the board')
print('Finished checking the board')
```

عند تشغيل البرنامج، ستظهر النتيجة الآتية على شاشة الحاسوب:



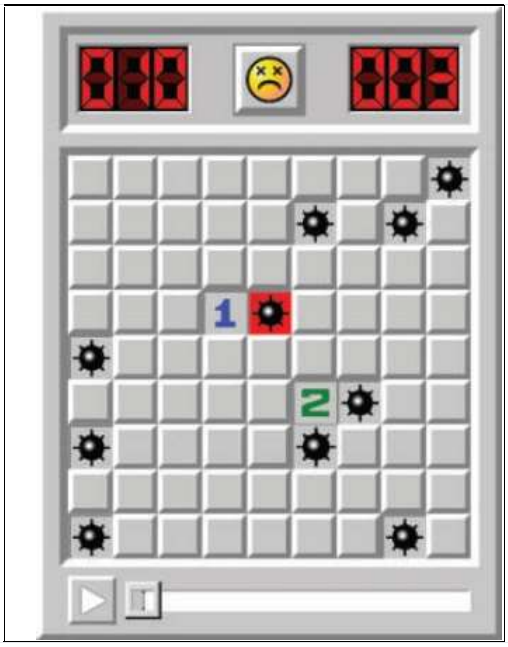
صحيح أنَّ التعامل مع القوائم المُركَّبة يتطلَّب كثيرًا استخدام حلقات تكرار مُركَّبة، لكنَّ ذلك ليس شرطًا.

تطبيقات عملية:

سنستعرض الآن مثالين على كيفية المرور على القوائم المُركَّبة، وهما يُمثِّلان جزءًا من بعض الألعاب التي تُمارَس على رقعة يُمكِن تمثيلها في صورة

قائمة مُركَّبة.

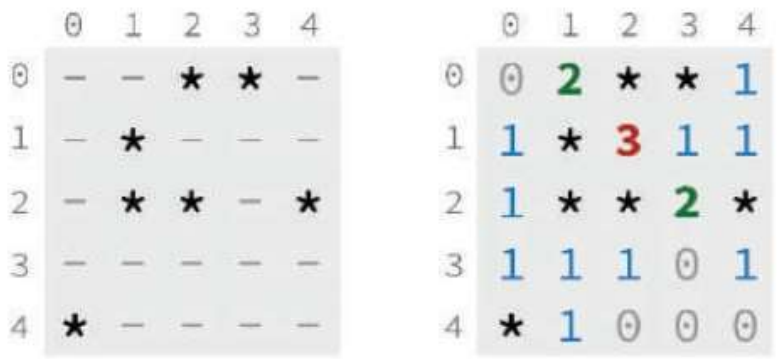
مثال:



تُعَدُّ لعبة كاسحة الألغام (Minesweeper) واحدة من الألعاب الكلاسيكية القديمة. وتقوم فكرة هذه اللعبة على محاولة تجنُّب الضغط على مُربَّع يحوي قنبلة؛ إذ تُقدِّم اللعبة تلميحات - عند الضغط على بعض المُربَّعات- عن عدد القنابل المَخْفية حول المُربَّع الذي يضغط عليه اللاعب.

في هذا المثال، لن نكتب برنامجًا كاملًا لهذه اللعبة، وإنَّما سنكتفي بمحاكاة عملية العَدِّ لعدد القنابل الموجودة حول كل مُربَّع، وتخزين هذا العدد في المُربَّع نفسه.

لنفترض أنَّ اللعبة مُمثَّلة بقائمة مُركَّبة، تحوي في كل عنصر علامة '-' في حال عدم وجود قنبلة، أو علامة '*' في حال وجود قنبلة. ووظيفتنا في هذا البرنامج هي استبدال رقم يُمثِّل عدد القنابل المُلاصِقة لذلك المُربَّع (مراعين المُربَّعات التي على يمين المُربَّع المطلوب، والمُربَّعات التي على شماله، والمُربَّعات التي فوقه، والمُربَّعات التي تحته مباشرة) بكل علامة '-'، أنظر الشكل (5-10)



	0	1	2	3	4
0	–	–	*	*	–
1	–	*	–	–	–
2	–	*	*	–	*
3	–	–	–	–	–
4	*	–	–	–	–

	0	1	2	3	4
0	0	2	*	*	1
1	1	*	3	1	1
2	1	*	*	2	*
3	1	1	1	0	1
4	*	1	0	0	0

الشكل (5–10): صورة القائمة قبل تنفيذ البرنامج وبعد تنفيذه.

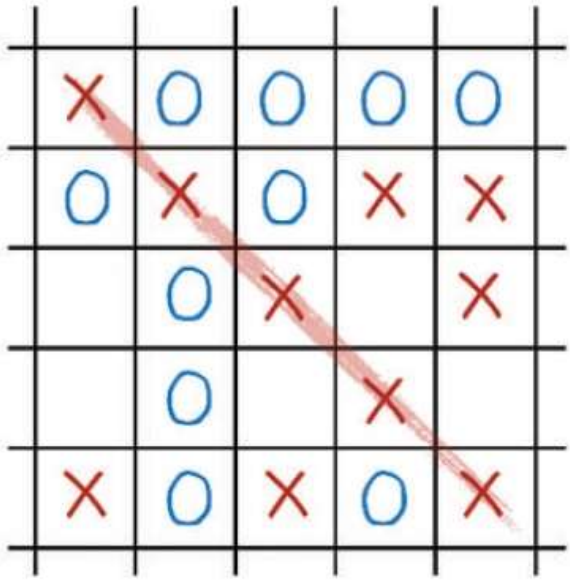
يكتفي البرنامج بالمرور على كل موقع في القائمة المُركَّبة، وعَدِّ القنابل التي حوله؛ أيْ إنَّه لن يمرَّ على كل عنصر في هذه الأثناء؛ لأنَّ الهدف ليس فقط معرفة قيمة العنصر، وإنَّما معرفة قيمة العناصر التي حوله أيضًا. ومن ثَمَّ يجب معرفة موقع العنصر؛ للتأكُّد من القِيَم المُخزَّنة في المواقع التي حوله، أنظر الشكل (5- 11)الذي يُبيِّن كيف يُنفَّذ المطلوب بافتراض أنَّ القائمة المُركَّبة تُسمّى(board)

```
board = [
['-', '*', '-', '-'],
['-', '-', '*', '-'],
['*', '-', '-', '-'],
['-', '*', '-', '*']
]
for i in range(len(board)):
    for j in range(len(board[i])):
        if board[i][j] == '-':
            count = 0
```

```
            if i > 0 and board[i-1][j] == '*':
                count += 1
            if i < len(board) - 1 and board[i+1][j] == '*':
                count += 1
            if j > 0 and board[i][j-1] == '*':
                count += 1
            if j < len(board[i]) - 1 and board[i][j+1] == '*':
                count += 1
            board[i][j] = str(count)
for row in board:
    line = ' '.join(row)
    print(line)
```

الشكل (5- 11): صورة القائمة المُركَّبة للعبة كاسحة الألغام.

مثال:



واحدة من الألعاب المشهورة التي تُلعَب عادةً باستخدام (XO) تُعَدُّ لعبة ، ويُمكِن ممارستها أيضًا باستخدام رقعة مُربَّعة بِغَضِّ 3x3رقعة حجمها 3 النظر عن حجمها.

في هذا المثال، لن نكتب برنامجًا كاملًا لهذه اللعبة، وإنَّما سنكتفي بكتابة الجزئية التي تتحقَّق من فوز أحد اللاعبين، وهو ما يتطلَّب المرور على ثلاثة أشياء، هي:

- كل صف.
- كل عمود.
- القُطْران.

في كل صف؛ (O) وعدد (X) سنبدأ أوَّلًا بالصفوف، ونتحقَّق من عدد فإذا كان العدد لأيٍّ منهما مساويًا لطول الصف، عَلِمْنا أنَّ أحد اللاعبين قد فاز.

```
# عدد الصفوف وهو نفسه عدد الأعمدة لأن الرقعة مربعة
N = len(board)
N = len(board)
for row in board:
    if row.count('X') == N:
        print('X wins!')
        break
    if row.count('O') == N:
        print('O wins!')
    break
```

أمّا بالنسبة إلى الأعمدة، فإنَّ الأمر مختلف بعض الشيء. فكل عمود يحوي عناصر من صفوف مختلفة؛ ما يتطلَّب المرور على كل عمود بصورة يدوية يمثل كل صف (i) يمثل موقع كل عمود و (j) حيث:

```
for j in range(N):
    # 1
    countX = 0
    countY = 0


for j in range(N):
    countX = 0
    countY = 0
    for i in range(N):
        if board[i][j] == 'X':
            countX += 1
        elif board[i][j] == 'O':
            countY += 1
        if countX == N:
            print('X wins')
            break
        if countY == N:
            print('Y wins')
            break
```

:، هي(j) في هذا المثال، يؤدّي المقطع البرمجي ثلاث مهام في كل عمود (O)، والآخر لحرف(X) تصفير عدّادين؛ أحدهما لحرف.1
) لعَدِّ مَرّات تكرار حرف)j (المرور على جميع العناصر في العمود .2 في ذلك العمود، وذلك باستخدام حلقة تكرار تمرُّ على (O) وحرف (X .في تلك القائمة (j) جميع الصفوف في القائمة المُركَّبة، وتتحقَّق من العنصر
أنَّ العدد مساوٍ - (j) التأكُّد - بعد الانتهاء من عَدِّ الأحرف في العمود .3 .لطول العمود، وهو ما يعني أنَّ أحد اللاعبين قد فاز

وأمّا القُطْران فيُمكِننا التحقُّق منهما باستخدام حلقة تكرار تعمل على تغيير الصف والعمود في كل دورة؛ إذ يبدأ القُطْر الأوَّل عند [0][0]، ثمَّ ينتقل إلى [1][1]، ثمَّ ينتقل إلى [2][2]، وهكذا

```
countX = 0
countY = 0
for i in range(N):
      if board[i][N-i-1] == 'X':
            countX += 1
      elif board[i][N-i-1] == 'O':
            countY += 1
if countX == N:
      print('X wins')
if countY == N:
      print('Y wins')
```

، ثمَّ [1][N-2] ، ثمَّ ينتقل إلى[0][N-1] في حين يبدأ القُطْر الثاني عند [2][N-3] ينتقل إلى، وهكذا.

```
countX = 0
countY = 0
for i in range(N):
        if board[i][N-i-1] == 'X':
                countX += 1
        elif board[i][N-i-1] == 'O':
                countY += 1
if countX == N:
        print('X wins')
if countY == N:
        print('Y wins')
```

المواطنةُ الرقميةُ:

- **الأخلاقيات الرقمية**: أحترم آراء الآخرين وأفكارهم عند مناقشة التعليمات البرمجية أو مناقشة المشروعات، وأُقدِّم النقد البَنّاء والمساعدة للآخرين عند مراجعة تعليمات البرمجية.
- **الوعي بالأمن السـيبراني**: أُدرِك أهمية اسـتخدام برامج مكافحة الفيروسات وتحديث أنظمة التشغيل بانتظام أثناء العمل في بيئة بايثون (Python).