5 STRUCTURED TYPES, MUTABILITY, AND HIGHER-ORDER FUNCTIONS

الانواع المركبة – الأنواع القابلة للتغير – الدوال عالية الرتبة

حتى الان، كل أنواع البرامج التى عملنا عليها كانت تحتوى على ثلاثة انواع أساسية من البيانات ، الأعداد الطبيعية INT والأعداد التى تحتوى على نقطة عائمة FLOAT والسلاسل الحرفية (النصوص) STR ، وكلها بيانات من النوع المفرد (Scalar) ، ولهذا يمكننا أن نقول أنها كائنات لا نستطيع الوصول الى مكوناتها لانها لا تحتوى على مكونات ، كل كائن منها عبارة عن وحدة واحدة .

على عكس النصوص (str) يمكننا أن نراها على أنها مركبة ، أو نوع من البيانات الغير فردية ، لانك قد تستخدم الفهرسة (indexing) لاستخراج حرف أو أكثر عن طريق تشريح النص وتكوين نصوص فرعية منه.

فى هذا الفصل ، سوف نتعرض الى ثلاثة أنواع جديدة للبيانات يمكننا أن نقول عنها متراكبة ، أى تتكون من أجزاء ، أولها يسمى المجموعة (tuble) وهو بالاحرى تعميم بسيط للنوع المعروف str . النوعين الآخرين أحدهما هى القائمة List والآخر هو الاملاءى (dict) ، وسوف تكون دراسها أكثر متعة باذن من الله. وذلك بسبب قابليتها للتغير.

سوف نعود أيضا للجزئية المتعلقة بالدوال ، مع بعض الامثلة التى تصور الادوات التى تتيح لنا التعامل مع الدوال مثلها مثل باقى انواع الكائنات.

5 - 1 المجموعات Tuples

مثل النصوص ، المجموعة تتكون من تتابع مرتب من العناصر. الفرق هنا هو ان عناصر المجموعة لا يشترط أن تكون حروف أو نصوص ، كل عنصر يمكن أن يكون أى نوع من البيانات ، ولا نحتاج الى ان تكون كل العناصر من نفس النوع ، بل ان كل عنصر يمكنه أن يكون نوع مستقل عن الاخر ، حرفيا تتم كتابة عناصر المجموعة كقائمة ويفصل بين كل عنصر وعنصر آخر بفاصلة ، ويحاط جميه العناصر بأقواس مدورة .على سبيل المثال يمكننا كتابة:

```
t1 = ( )
t2 = (1, 'two', 3)
print t1
print t2
```

ولا داعى للدهشة عندما ترى أن أمر الطباعة ينتج لك المخرجات التالية :

```
( )
(1, 'two', 3)
```

بالنظر الى هذا المثال ، ربما – و هذا شىء طبيعى – تعتقد ان المجموعة تحتوى على قيمة مفردة ، 1 سوف يتم كتابتها (1) ولكن ، على حد تعبير ريتشارد نيكسون (that would be wrong) .

عندما ترى التعبير (1) ربما تعتقد ان هذه طريقة مطولة لكتابة العدد الصحيح 1 ، ولكنها تكتب هكذا للدلالة على مجموعة مفردات تحتوي على هذه القيمة ، نكتب (1,) ، تقريبا كل شخص يستخدم بايثون لمرة أو أكثر قد قام بحذف - عن طريق الخطأ - تلك الفاصلة المزعجة ، مثل السلاسل النصية ، يمكن تجميع المجموعات ، فهرستها ، وتسريحها .انظر :

```
t1 = (1, 'two', 3)
t2 = (t1, 3.25)
print t2
print (t1 + t2)
print (t1 + t2)[3]
print (t1 + t2)[2:5]
```

التعيين الُثانى يربط المتغير t2 الى القيم التالية : المجموعة التى تم تعيينها الى المتغير t1 بالاضافة الى الرقم العشرى 3.25 ، وهذا جائز ، لأنه مثله مثل أى شىء آخر فى بايثون ، عبارة عن كائن ، لذا فالمجموعة يمكن أن تحتوى بداخلها على مجموعات ، تماما كما أخرجته لنا أمر الطباعة الأول :

```
((1, 'two', 3), 3.25)
```

أخرج لنا أمر الطباعة الثانى القيمة التى تم توليدها عن طريق الجمع بين القيمة التى تم تعيينها الى المتغير t1 مضافا اليها القيمة التى تم تعيينها الى المتغير t2 وقامت باخراج الناتج:

```
(1, 'two', 3, (1, 'two', 3), 3.25)
```

امر الطباعة التالى قام بانتقاء وطباعة العنصر الرابع الناتج من جمع المجموعات (دائما ما تبدأ الفهارس فى بايثون من الصفر) ، وقامم أمر الطباعة التالى بانشاء وطباعة شريحة من المجموعة كما فى المخرجات التالية :

```
(1, 'two', 3)
(3, (1, 'two', 3), 3.25)
```

يمكننا استخدام الحلقة التكرارية for لتكرر نفسها وتلتقط عناصر المجموعة ، على سبيل المثال الكود التالى يقوم بحساب القواسم المشتركة للأعداد 20 و 100 ثم يقوم بعدها بجمع هذه القواسم :

```
def findDivisors (n1, n2):
    """ Assumes that n1 and n2 are positive ints
        Returns a tuple containing all common divisors of n1 & n2"""
      divisors = () #the empty tuple
      for i in range(1, min (n1, n2) + 1):
       if n1%i == 0 and n2%i == 0:
          divisors = divisors + (i,)
    return divisors
divisors = findDivisors(20, 100)
print divisors total = 0
for d in divisors:
    total += d
print total
```

5 - 1 - 1 التتالى والتعيين المتعدد Sequences and Multiple Assignment

اذا كنت تعرف عدد مكونات سياق معين – على سبيل المثال سلسلة حرفية أو مجموعة – انه الوقت المناسب لاستخدام التعيين المتعدد فى بايثون لاستخراج عناصر مفردة ، على سبيل المثال ، بعد تنفيذ حالة التعيين (x , y = (3, 4 سوف يتم تعيين القيمة 3 الى x وتعيين القيمة 4 الى y ، وبالمثل حالة التعيين a, b, c = 'xyz' سوف يتم تعيين القيمة 'x' الى المتغير a والقيمة 'y' الى المتغير b والقيمة 'z' الى المتغير c . هذه الآلية تناسب الدوال التى تعيد سياقات محددة الحجم.

```
Consider, for example the function
def findExtremeDivisors(n1, n2):
    """ Assumes that n1 and n2 are positive ints
        Returns a tuple containing the smallest common divisor > 1 and the
        largest common divisor of n1 and n2 """
    divisors = () #the empty tuple
    minVal, maxVal = None, None
    for i in range(2, min(n1, n2) + 1):
        if n1%i == 0 and n2%i == 0:
            if minVal == None or i < minVal:
                minVal = i
            if maxVal == None or i > maxVal:
                maxVal = i
    return (minVal, maxVal)



The multiple assignment statement
minDivisor, maxDivisor = findExtremeDivisors(100, 200) will bind
 minDivisor to 2 and maxDivisor to 100.
```

5 - 2 القوائم والقابلية للتغير Lists and Mutability

مثل المجموعات ، القواءم Lists هى سياق مرتب من القيم ، عندما تكون كل قيمة معرفة بفهرس . القاعدة اللغوية للتعبير عن عناصر القائمة يشبه تلك المستخدمة فى المجموعات ؛ الفرق هو أننا نستخدم أقواسا مربعة بدلا

من تلك الدائرية التى نستخدمها مع المجموعات ، والقائمة الفارغة تكتب [ ] ، والقائمة المفردة بدون تلك الفاصلة قبل اغلاق القوس ، لهذا ، على سبيل المثال :

```
L = ['I did it all', 4, 'love']
for i in range(len(L)):
    print L[i]
```

سوف تكون النتيجة :

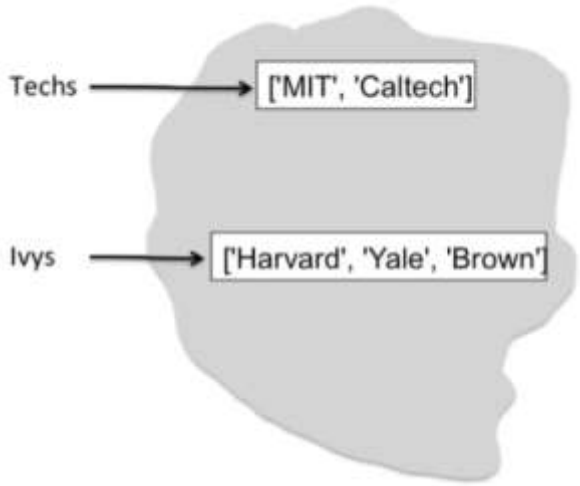
```
I did it all
4
love
```

بالمناسبة ، حقيقة استخدام الاقواس المربعة هى لأنه حرفيا نوع البيانات – القواءم Lists – مفهرسة الى عناصر ، وتشريح القوائم ربما يؤدى الى بعض التداخل أو التعارض ، على سبيل المثال ، التعبير [1][1:3][1, 2, 3, 4] ، قيمته 3 يستخدم الاقواس المربعة بثلاث طرق مختلفة ، هذه نادرا ما يكون بسبب مشكلة فى الممارسة ، لانه فى معظم الوقت يتم بناء القوائم عن طريق الاطراد وليس حرفيا كما يحدث فى المجموعات ، وتختلف القوائم عن المجموعات فى أن الأولى قابلة للتغير Mutable . في المقابل ، المجموعات و السلاسل الحرفية غير قابلة للتغيير Immutable . هناك العديد من العوامل التي يمكن استخدامها لإنشاء كائنات من هذه الأنواع الغير قابلة للتغيير . والمتغيرات يمكن ربطها بكائنات من هذه الأنواع . لكن الكائنات من النوع Immutable لا يمكن التعديل عليها . بينما على الجانب الآخر فان الكائنات من النوع List يمكن التعديل عليها بعد انشائها ، التمييز بين عملية تغير كائن وعملية تعيين كائن الى متغير ربما – للوهلة الاولى – نراها خفية . على كل حال يمكنك الاستمرار فى تلاوة التعويذة .

فى بايثون المتغير هو مجرد اسم ،ن على سبيل المثال – علامة يتم الحاقها بكائن ما ، ربما عملت التعويذة عملها وظهر جزء من الغموض ، لنرى هذه السطور ربما تتضح الامور أكثر :

```
Techs = ['MIT', 'Caltech']
Ivys = ['Harvard', 'Yale', 'Brown']
```

عند التنفيذ ، سيقوم المترجم Interpreter بانشاء قائمتين ، وربط كل واحدة بالاسم المناسب لها كما فى الشكل التالى :



الشكل 5 – 1 قائمتين

ولنلاحظأيضا فى عمليات التعيين الآتية :

```
Univs = [Techs, Ivys]
Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
```

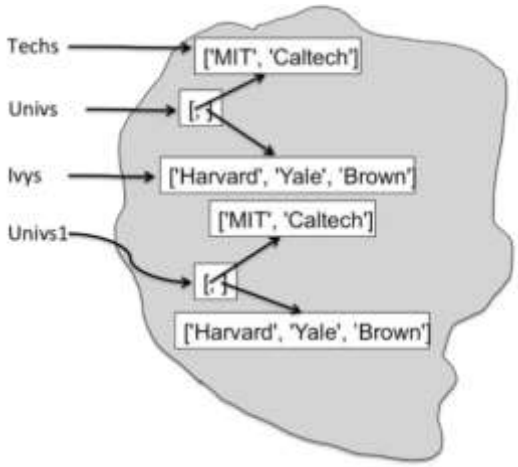
هنا أيضا يقوم المترجم بانشاء قائمتين جديدتين وبربط لهما المتغيرات كما يبدو لنا . وعناصر هذه القواءم هى نفسها قوائم ، لنأخذ أوامر الطباعة الثلاثة الآتية :

```
print 'Univs =', Univs
 print 'Univs1 =', Univs1
print Univs == Univs1
```

وشيكون الناتج :

```
Univs = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
True
```

ما هذا ! ، يبدو كما لو أنهما قائمة واحدة ، ويبدو أيضا أن المتغيرات univ و univ1 قد تم ربطهما الى نفس القائمة ، لكن قد تبدو المظاهر خادعة فى بعض الاحيان ، وهذا ما سوف يتضح فى الشكل التالى أن المتغيرات univ و univ1قد تم ربط كل منهما الى قيمة مغايرة تماما للاخرى :



الشكل 5 – 2 صورة لقائمتين يبدو أنها واحدة لكنهما ليستا كذلك

الصورة توضح أن المتغيرات univ و univ1 مربوطة الى قيم منفصلة تماما عن بعضها ، يمكننا التحقق من ذلك باستخدام الدالة الداخلية idوالتى تعيد رقم تعريفى مختلف وفريد لكل كائن مخزن فى الذاكرة . وهذه الدالة عادة ما تستخدم لاختبار التكافؤ ، عندما يتم اجراء الكود لاتالى :

```
print Univs == Univs1 #test value equality
print id(Univs) == id(Univs1) #test object equality
print 'Id of Univs =', id(Univs)
```

```
print 'Id of Univs1 =', id(Univs1)
```

وسيكون ناتج الطباعة

```
True
False
 Id of Univs = 24499264
Id of Univs1 = 24500504
```

لا تتوقع أن تحصل على نفس المعرفات عند اجراء هذا الكود ، الدلالات اللفظية فى بايثون semanticsتقول لا تسأل عن ماهية معرفات الكائنات ولا كيف يتم توليدها ، هى فقط لمجرد تحقيق ان يكون لكل كائن رقم تعريفى فريد ،لاحظ فى الشكل 5 – 2 ان عناصر univ هى ليست نسخ من عناصر المجموعة المربوط اليها techs and lvys ولكن بالأحرى فانها القوائم نفسها ، العناصر فى univ1 هى قوائم تحتوى على نفس العناصر والتى تم سردها فى القائمة univ . لكنها ليست العناصر نفسها ، يمكننا التحقق من ذلك باجراء الكود :

```
print 'Ids of Univs[0] and Univs[1]', id(Univs[0]), id(Univs[1])
print 'Ids of Univs1[0] and Univs1[1]', id(Univs1[0]), id(Univs1[1])
```

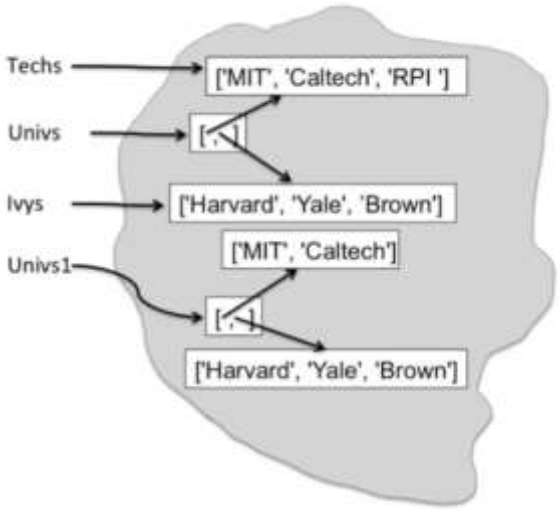
وستكون النتيجة :

```
Ids of Univs[0] and Univs[1] 22287944 22286464
Ids of Univs1[0] and Univs1[1] 22184184 22287984
```

لماذا يحدث هذا؟! .. سبب هذا أن القوائم Lists هى كائنات قابلة للتغير ، وقد نلاحظ ذلك بعد اجراء :

```
Techs.append('RPI')
```

الأسلوب append له جانب تأثيرى ، فبدلا من انشاء قائمة جديدة ، هى تقوم بالتعديلفى القائمة الموجودة بالفعل Techs عن طريق اضافة عنصر جديد ، وسيتم اضافة النص 'RPI' بعد آخر العناصر الموجودة . وبعد الحاق العنصر ستكون حالى الحوسبة كالتالى :



الشكل 5 – 3 الاثبات لعملية التحور

القائمة univ لازالت تحتوى على نفس القائمتين ، ولكن محتوى هاتين القائمتين قد تغيرتا ، فلنقم بإجراء أمر الطباعة التالى :

```
print 'Univs =', Univs
print 'Univs1 =', Univs1
```

وستكون نتيجة الطباعة :

```
Univs = [['MIT', 'Caltech', 'RPI'], ['Harvard', 'Yale', 'Brown']]
Univs1 = [['MIT', 'Caltech'], ['Harvard', 'Yale', 'Brown']]
```

ما حدث لدينا هنا هو شىء يسمى Aliasing – الاسم المستعار - ، يوجد هناك مساران متمايزان ، مسار واحد من خلال المتغير Tech والآخر من خلال العنصر الأول لكائن القائمة الذي يرتبط به Univ ، أحدهم أن يحور الكائن عبر أي من المسارين ، وسوف يكون تأثير التحور مرئياً من خلال كلا المسارين . قد يكون هذا مناسبا ، لكنه أيضا قد يكون غادرا . تؤدى الأسماء المستعارة الغير مقصودة الى أخطاء برمجية يكون من الصعب تعقبها .

كما فى المجموعات، الحلقة التكرارية for يمكنها أن تدور داخل القائمة وتخرج لنا محتواها ،

```
for e in Univ:
    print 'Univ contains', e
    print '  which contains'
    for u in e:
        print '    ', u
```

وسيكون الناتج :

```
Univs contains ['MIT', 'Caltech', 'RPI']
  which contains
    MIT
    Caltech
    RPI
Univs contains ['Harvard', 'Yale', 'Brown']
  which contains
    Harvard
    Yale
    Brown
```

عندما نضيف قائمة إلى أخرى ، على سبيل المثال ، (Techs.append (Ivys)) ، يتم الحفاظ على البنية الأصلية ، والنتيجة هي قائمة تحتوي على قائمة. لنفترض أننا لا نريد الحفاظ على هذا الهيكل ، ولكن نريد إضافة عناصر قائمة واحدة إلى قائمة أخرى. يمكننا القيام بذلك عن طريق استخدام تسلسل القوائم Concatenate أو طريقة التمديد Extend ، على سبيل المثال ،

```
L1 = [1,2,3]
L2 = [4,5,6]
L3 = L1 + L2
print 'L3 =', L3
L1.extend(L2)
```

```
print 'L1 =', L1 L1.append(L2)
print 'L1 =', L1
```

وسيكون الناتج :

```
L3 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6]
L1 = [1, 2, 3, 4, 5, 6, [4, 5, 6]
```

لاحظ أن المشغل + ليس له تأثير جانبي. يقوم بإنشاء قائمة جديدة وإرجاعها. في المقابل ، وتمديد وإلحاق كل تحور الى القائمة L1.

يحتوي الشكل 5.4 على أوصاف مختصرة لبعض الطرق المرتبطة بالقوائم. لاحظ أن كل هذه الأرقام باستثناء العد والفهرس تحوّل القائمة

```
L.append(e) adds the object e to the end of L.
L.count(e) returns the number of times that e occurs in L.
L.insert(i, e) inserts the object e into L at index i.
L.extend(L1) adds the items in list L1 to the end of L.
L.remove(e) deletes the first occurrence of e from L.
L.index(e) returns the index of the first occurrence of e in L.  It raises an exception (see Chapter 7) if e is not in L.
L.pop(i) removes and returns the item at index i in L.  If i is omitted, it defaults to -1, to remove and return the last element of L.
L.sort() sorts the elements of L in ascending order.
L.reverse() reverses the order of the elements in L.
```

5 – 2 – 1 النسخ Cloning

على الرغم من أنه متاح ، لكن تجنب التتغيير فى قائمة تخضع للتكرار، لاحظ المثال التالى :

```
def removeDups(L1, L2):
    """Assumes that L1 and L2 are lists.
      Removes any element from L1 that also occurs in L2"""
    for e1 in L1:
       if e1 in L2:
          L1.remove(e1) L1 = [1,2,3,4] L2 = [1,2,5,6]
removeDups(L1, L2)
print 'L1 =', L1
```

ربما تتفاجأ عندما ترى أمر الطباعة يخرج هذه النتيجة :

```
L1 = [2, 3, 4]
```

خلال الحلقة for ، يقوم التطبيق Python Implementation بتتبع مكان وجوده في القائمة باستخدام عداد داخلي يتزايد في نهاية كل عملية تكرار. عندما تصل قيمة العداد إلى الطول الحالي للقائمة تتوقف الحلقة ، هذا قد يؤدى كما قد نتوقع منه أن يفعل إذا لم يتم تحور القائمة داخل الحلقة التكرارية ، ولكن يمكن أن يكون لها عواقب مفاجئة إذا تم تحوير القائمة ، في هذه الحالة ، يبدأ العداد المخفي عند 0 ، ويكتشف أن L1[0] في L2 ، ويزيله - مما يقلل من طول L1 إلى 3. ثم يتم زيادة العداد إلى 1 ، وتستمر عملية للتحقق مما إذا كانت قيمة L1[1] موجودة في L2 . لاحظ أن القيمة الحالية لـ L1[1] ليست 2 ولكنها أصبحت 3 ، كما ترى أنه من السهل اكتشاف ما يحدث . . . تبدو أنها غير مقصودة كما فى هذا المثال .

ولتجنب هذا النوع من المشاكل التى قد تنتج بغير قصد .. يفضل استخدام النسخ Clone لاجراء التقطيع أو تشريح القوائم ، (بمعنى انشاء نسخة من) القائمة وكتابتها فى e1 فى القائمة L1[:] .

و لاحظ انه عن طريق كتابة newL1 = L1 متبوعا بـ for e1 in newL1 لم يتم حل المشكلة لاننا بهذه الطريقة لم ننشئ نسخة جديدة من L1بل قمنا بتقديم نفس القائمة باسم جديد .

طريقة التقطيع slice ليست هى الطريقة الوحيدة لانشاء جديدة من الكائنات فى بايثون ، التعبير list(L) يعيد لنا نسخة جديدة من L ، واذا كانت القائمة تحتوى على كائنات قابلة للتغير وتريد استنساخها فيمكنك استدعاء المكتبة copy واستخدام الدالة copy.deepCopy

5 – 2 – 2 فهم القائمة List Comprehension

.

يقدم لنا فهم القائمة طريقة مختصرة لتطبيق عملية ما على العناصر فى سياق القائمة ، ويقوم بانشاء قائمة جديدة تكون عناصرها هى العناصر التى تم تطبيق العمليات عليها ، على سبيل المثال انظر العناصر فى القائمة التالية :

```
L = [x**2 for x in range(1,7)] print L
will print the list
[1, 4, 9, 16, 25, 36]
```

التكرار فى الحلقة التكرارية for يمكن أن يتعبه حالة شرطية if أو تكرارية for ، واحدة أو أكثر ، والتى يتم تطبيقها على القيم التى ينتجها هذا التكرار . هذه الحلقات التكرارية الاضافية التى تم تطبيقها تقوم بتعديل القيم التى أنتجتها الحلقة التكرارية الاولى ، وتنتج سياقا جديدا من القيم ، الى أى العمليات المرتبطة يتم التطبيق ..

هذا المثال سوف يوضح أكثر :

```
mixed = [1, 2, 'a', 3, 4.0]
print [x**2 for x in mixed if type(x) == int]
```

سوف يقوم بتربيع القيم فى السياق mixed فقط اذا كانت أعداد صحيحة ويطبع : [1, 4, 9]

بعض مبرمجين بايثون يحبون استخدام فهم القوائم List Comprehension بعدة طرق قد تكون رائعة ومبهمة ، لكن تذكر أن أناس آخرون قد يقرءون أكوادك ، والإبهام ليست خاصية مرغوبة .

5 – 3 الدوال ككائنات Functions as Objects

فى البايثون ، الدوال functions هى فى المقام الاول كائنات Objects ، وهذا يعنى أنه يمكن معاملته على أنها كائنات من نوع ما ، على سبيل المثال ، الأعداد الصحيحة Int أو القوائم List ، لها أنواع Types ، فمثلا ، التعبير type(fact) سيكون له القيمة < type 'function' > ؛ يمكنها أيضا أن تظهر فى التعبيرات ، فمثلا ، فى الجانب الأيمن من عمليات التعيين ، أو تعيين متغير لدالة ، أو يمكنها أن تكون عنصر من عناصر قائمة .

استخدام الدوال كمتغيرات Arguments يمكن أن يكون مناسب بشكل جزئى عند الاقتران بالقوائم Lists ، وهى تقودنا الى اسلوب فى البرمجة يسمى البرمجة عالية الرتبة Higher-Order-programming ، ناخذ مثالا على ذلك :

```
def applyToEach(L, f):
    """Assumes L is a list, f a function
       Mutates L by replacing each element, e, of L by f(e)"""
    for i in range(len(L)):
        L[i] = f(L[i])

L = [1, -2, 3.33]
print 'L =', L
print 'Apply abs to each element of L.' applyToEach(L, abs)
print 'L =', L
print 'Apply int to each element of', L applyToEach(L, int)
print 'L =', L
print 'Apply factorial to each element of', L applyToEach(L, factR)
print 'L =', L
print 'Apply Fibonnaci to each element of', L applyToEach(L, fib)
```

الشكل 5-5 تطبيق الدالة على كل عنصر من عناصر القائمة

الدالة التى يتم تطبيقها على الكل تسمى عالية الرتبة Higher-Order ، وذلك لانها لا تحتوى على متغيرات ، هى بذاتها دالة ، عند تطبيقها فى أول سطر من الكود الموجود فى الشكل 5-5 ، قامت بتغيير عناصر L عن طريق تطبيق الدالة abs الى كل عنصر من عناصرها ، وفى المرة الثانية قامت بتطبيق الدالة Int الى كل العناصر فقامت باجراء تحويل للانواع . . . وهكذا ....

```
L = [1, -2, 3.3300000000000001]
Apply abs to each element of L.
```

```
L = [1, 2, 3.3300000000000001]
Apply int to each element of [1, 2, 3.3300000000000001]
L = [1, 2, 3]
Apply factorial to each element of [1, 2, 3]
L = [1, 2, 6]
Apply Fibonnaci to each element of [1, 2, 6]
L = [1, 2, 13]
```

البايثون تحتوى على دوال من النوع عالى الرتبة مثل map والتى تشبه هذه الدالة لكنها أكثر شمولية ، فلتعيين كل دالة الى كل عناصر القائمة L كما فى الشكل 5-5 يمكننا استخدام الدالة map فى أبسط اشكالها ، وهو ما يسمى الدالة الوحيدة (الدالة التى تحتوى على متغير وحيد) والمتغير الثانى عبارة عن أى ترتيب من المتغيرات التى يمكن ان تكون من أى نوع من القيم ، وتعيد لنا قائمةتم توليدها عن طريق تطبيق الدالة فى المتغير الاول الى كل عنصر من عناصر المتغير الثانى ، فمثلا ، التعبير map(factR, [1, 2, 3]) يعيد لنا القائمة [1, 2, 6] .

عامة ، المتغير الاول للدالة map يمكنه أن يكون دالة بأى عدد n من المتغيرات ، والتى من المفترض ان تتوافق مع عدد التتابع فى السياق الموجود فى ترتيب المتغيرات التالية له ،

النصوص – المجموعات - القوائم Strings – Tuples - Lists

لقدرأينا ثلاثة أنواعع مختلفة من أنواع البيانات ، وتجمعها خصائص مشتركة فى أنه لأى كائن من هذه الأنواع يمكن تطبيق الدوال المبينة فى الشكل التالى :

```
seq[i] returns the ith element in the sequence.
len(seq) returns the length of the sequence.
seq1 + seq2 returns the concatenation of the two sequences.
n * seq returns a sequence that repeats seq n times.
seq[start:end] returns a slice of the sequence.
e in seq is True if e is contained in the sequence and False otherwise.
 e not in seq is True if e is not in the sequence and False otherwise.
 for e in seq iterates over the elements of the sequence
```

الشكل 5-6 العمليات المشتركة لأنواع السياق

وأيضا مقارنة لبعض الاشياء المتشابهة والمختلفة فى كل نوع مختصرة كما فى الشكل التالى :

النوع	نوع العناصر	أمثلة لأنواع العناصر	القابلية للتغير
str	Characters	' ' , ' a ' , ' abcd ' , ' any string'	immutable
tuple	Any	( ), (1), (2, 'a', 'abcd'), ('a', (2, 3))	immutable
list	Any	[ ], [3], ['abc', 4]	mutable

الشكل 5-7 مقارنة بين أنواع السياقات

يميل مبرمجى البايثون الى استخدام القوائم lists اكثر من استخدامهم للمجموعات tuples ، وذلك لانها قابلة للتغير ، ولانها أيضا يمكن اضافة أى عدد من العناصر اليها بشكل مطرد .

```
evenElems = [ ]
for e in L:
  if e%2 == 0:
     evenElems.append(e)
```

أحد الخواص التى تجعل المجموعات مفضلة هى كونها غير قابلة للتغير ، الاسماء البديلة ليست مقلقة ، وعدم قابليتها للتغير يجعل منها الحل الافضل لكى تستخدم كمفاتيح فى قاموس كما سنرى فى القسم التالى.

لأن النصوص strings تحتوى على أحرف فقط ، ربما هذا يجعلها أقل استخداما فى البرمجة ، لكن على الجانب الاخر نجد العديد من الدوال الداخلية فى اللغة والتى تجعل الحياة أكثر سهوله عند التعامل مع النصوص ، سيكون هذا موضحا باختصار فى الجدول التالى :

الدالة	الوصف
s.count(s1)	تحسب كم مرة تكرر النص s1 بداخل النص s
s.find(s1)	تعيد رقم الفهرس لاول حرف من النص الفرعى s1 اذا كان موجودا داخل النص s أو تعيد القيمة -1 اذا لم يتم العثور على النص الفرعى
s.rfind(s1)	نفي الدالة السابقة ولكنها تعمل بشكل عكسى أى تبدأ من النهاية (الحرف r يشير الى reverse)
s.index(s1)	نفس الدالة find ولكنها تعيد حالة اعتراض(انظر الفصل السابع) exception فى حالة

	عدم العثور على النص الفرعى
s.rfind(s1)	نفس الدالة index ولكنها تعمل بشكل عكسى
s.lower( )	تبدل حالى كل الأحرف الى smallCase
s.replace(old, new)	تقوم باستبدال كل النصوص الفرعية التى توافق القيمة old بالقمة الجديدة new
s.rstrip( )	تقوم هذه الدالة بحذف المساحات البيضاء
s.split(d)	يستطيع تقسيم النص على أساس محدد delimiter ويعيد قائمة تحتوى على النصوص المقطّعة واذا غاب المتغير d يتم التقسيم على أساس محددات افتراضية وجدث مثل (space, tab, newline, return, formFeed)

الشكل 5-8 بعض الاساليب methods الخاصة بالنصوص

5 – 5 القواميس Dictionaries

الكائنات من النوع Dict (اختصارا لـ dictionary) تشبه القوائم غير أن الفهارس (Indices) لا يشترط أن تكون أرقام صحيحة .. بل يمكنها ان تكون أى نوع من الانواع الغير قابلة للغير ، ولأنها غير مرتبة سنسميها مفاتيح ، يمكنك أيضاً النظر الى الكائنات من النوع dict على انها أزواج من (المفاتيح / القيم) أو (Keys / Values) .

حرفيا يتم تقديم الكائنات من النوع Dict محاطة بأقواس مجعدة {} او curly braces ويتم ادراج العناصر بالداخل على هيئة (مفتاح : قيمة) أى كلمة تعبر عن المفتاح متبوعة بنقطتين رأسيتين (: colon) ثم قيمة هذا المفتاح .

المثال التالى يوضح الطريقة المتبعة لتعريف كائن من النوع dict :

```
monthNumbers = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5,
                1:'Jan', 2:'Feb', 3:'Mar', 4:'Apr', 5:'May'}
print 'The fourth month is ' + monthNumbers[3]
dist = monthNumbers['Apr'] - monthNumbers['Jan']
print 'Apr and Jan are', dist, 'months apart'
```

وستكون النتيجة :

```
The fourth month is Mar Apr
and Jan are 3 months apart
```

المدخلات فى القاموس غير مرتبة ويصعب استهداف البيانات عن طريق الفهارس ، ففى المثال السابق وعند طلب الرقم الخاص بشهر يناير بهذا الشكل monthNumber[1] سيعيد لنا نتيجة غير التى نرغب بها ، فالفهرس 1 يشير الى فبراير وليس يناير ، ويمكننا معرفة ذلك بشكل أكثر قربا عندما نستخدم الوسيلة keys والتى تعيد لنا قائمة بداخلها كل المفاتيح الموجودة فى الكائن من النوع dict ، فاذا طلبنا منه الامر monthNumber.keys() ربما يكون العائد شيئا من هذا القبيل ['Feb', 5, 'Apr', 'Jan', 'May', 3, 4] . [1, 2, Mar'

عند استخدام الحلقة التكرارية for فان القيم التى يتم اضافتها الى القاءمة المتولدة هى عبارى عن مفاتيح هذا القاموس وليس (المفتاح / قيمته) كما فى الكود التالى :

```
for e in monthNumbers:
    keys.append(e)
    keys.sort()
print keys
prints [1, 2, 3, 4, 5, 'Apr', 'Feb', 'Jan', 'Mar', 'May']
```

الكائنات من النوع dic هى إضافة عظيمة للغة البايثون ، فهى تجعل كتابة العديد من البرامج شيئا سهلا ، على سبيل المثال وفى الشكل التالى كتابة قاموس يستخدم فى الترجمة بين اللغات ليس بالشىء الصعب :

```
# تمثيل لقاموس مصغر لبعض الكلمات من الانجليزية الى الفرنسية
EtoF = {'bread':'pain', 'wine':'vin', 'with':'avec', 'I':'Je','eat':'mange',
'drink':'bois', 'John':'Jean',
            'friends':'amis', 'and': 'et', 'of':'du','red':'rouge'}
# تمثيل لقاموس مصغر لبعض الكلمات من الفرنسية الى الانجليزية
FtoE = {'pain':'bread', 'vin':'wine', 'avec':'with', 'Je':'I', 'mange':'eat',
'bois':'drink', 'Jean':'John',
 'amis':'friends', 'et':'and', 'du':'of', 'rouge':'red'}
# تمثيل لقاموس كبير يجمع كل القواميس التى تم تعريفها
dicts = {'English to French':EtoF, 'French to English':FtoE}

# التعريف لدالة الترجمة لكل كلمة على حدة
def translateWord(word, dictionary):
```

```
    if word in dictionary.keys():
 return dictionary[word]
    elif word != '':
 return '"' + word + '"'
    return word
# التعريف لدالة الترجمة لمقطع كامل
def translate(phrase, dicts, direction):
    UCLetters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
    LCLetters = 'abcdefghijklmnopqrstuvwxyz'
    letters = UCLetters + LCLetters
    dictionary = dicts[direction]
    translation = ''
    word = ''
   for c in phrase:
     if c in letters:
         word = word + c
       else:
         translation = translation+ translateWord(word, dictionary) + c
         word = ''
     return translation + ' ' + translateWord(word, dictionary)
print translate('I drink good red wine, and eat bread.', dicts, 'English to
French')
print translate('Je bois du vin rouge.', dicts, 'French to English')
```

الشكل 5-9 الترجمة الحرفية

وستكون نتيجة الترجمة كالتالى :

```
Je bois "good" rouge vin, et mange pain.
I drink of wine red.
```

كما ترى ، ومثل القوائم ، فان الكائنات dict هى كائنات muable أى قابلة للتغير ، فلابد أن تكون حذرا فيما يخص الجانب التأثيرى ، لاحظ المثال :

```
FtoE['bois'] = 'wood'
print translate('Je bois du vin rouge.', dicts, 'French to English' )
        = = =    will print      = = =
I wood of wine red
```

لقد أضفنا فهرس الى القاموس بكود مثل هذا : FtoE['blanc'] = 'white' .

كالقوائم ، هناك عدة وسائل method للتعامل مع الكائنات من النوع Dict سنعرض لها فيما يتناسب مع الامثلة فى مكان متقدم من هذا الكتاب ، ونقدم منها على بيل المثال لا الحصر :

الدالة	الوصف
Len(d)	تحسب عدد عناصر القاموس d
d.keys()	تعيد قائمة تكون عناصرها المفاتيح المتاحة أو الموجودة فى الفهرس d
d.values	تعيد قائمة تكون عناصرها القيم الموجودة فى القاموس d
K in d	تعيد True اذا كانت k موجودة فى d
d[k]	تعيد القيمة المقابلة للمفتاح k
d.get(k, v)	تعيد القيمة d[k] اذا كانت موجودة أو v اذا كان العكس
d[k] = v	تقوم بتعيين القيمة v داخل القاموس الى المفتاح k اذا كان موجودا، أو تنشئه
Del d[k]	تقوم هذه الدالة بحذف المفتاح k مع القيمة المقابلة
For k in d	تقوم بالتكرار على كل المفاتيح الموجودة فى d وتطبق عليها العمليات المطلوبة

الشكل 5-10 بعض الاساليب المرتبطة بالبيانات من النوع DICT

معظم لغات البرمجة لا تحتوى على مثل هذا النوع من البيانات ، لكن المبرمجين يبتكرون حيلا لتفادى هذا ، فعلى سبيل المثال يمكن تسجيل القيم هكذا ، dict = (('word1', 'trans1'), ('word1', 'trans1'), ('word1', 'trans1')) ، ويمكن استهداف كلمة معينة داخل هذا الترتيب بتعديل سيناريو الطلب كأن تطلب من حلقة تكرارية من النوع for ان تكرر مقارنة على جميع العناصر فان وجدت القيمة التى تقابل الفهرس 0 من عنصر داخل عناصر القاموس التى سنعتبرها مفتاحا ستعيد القيمة التى تقابل الفهرس 1 ، هذا قد يكون حلا للمشكلة لكن تصور بدلا من أن تذهب للكلمة المستهدفة بشكل مباشر ستضطر الى تكرار الاختبار على كل العناصر الموجودة ، وسوف يتوقف وقت التنفيذ على حجم القاموس ، ولن تستطيع تحديد مكان الكلمة المستهدفة فى كل مرة فربما كان مكانها فى آخر القاموس .

على النقيض ، ستجد الدوال المبنية والجاهزة للتطبيق سريعة بشكل مذهل ، يعتمد مطوروا اللغة على ما يسمى بالتشفير hashing (سيتم شرحه فى الفصل العاشر) ، وهذا من شأنه تقليص وقت التنفيذ الى أقل ما يمكن .