

5 STRUCTURED TYPES, MUTABILITY, AND HIGHER-ORDER FUNCTIONS

الانواع المركبة – الأنواع القابلة للتغير – الدوال عالية الرتبة

حتى الان، كل أنواع البرامج التي عملنا عليها كانت تحتوى على ثلاثة أنواع أساسية من البيانات ، الأعداد الطبيعية INT والأعداد التي 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 لأنها 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 (tuple) 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 الآخر ، حرفيا تتم كتابة عناصر المجموعة كقائمة ويفصل بين كل عنصر وعنصر آخر بفاصلة ، ويحاط جميعه 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 شئ طبيعي - تعتقد ان المجموعة تحتوي 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 الثانى 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 شئ 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)
```

أمر الطباعة التالى قام بانتقاء وطباعة العنصر الرابع الناتج من جمع المجموعات (دائما ما تبدأ الفهارس فى بايثون من الصفر) ، وقامم أمر الطباعة التالى بإنشاء 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
```

إذا كنت تعرف عدد مكونات سياق معين – على سبيل المثال سلسلة حرفية أو مجموعة – أنه الوقت المناسب لاستخدام التعيين المتعدد في بايثون لاستخراج عناصر مفردة ، على سبيل المثال ، بعد تنفيذ حالة التعيين $x, y = (3, 4)$ ، سوف يتم تعيين القيمة 3 إلى x وتعيين القيمة 4 إلى y ، وبالمثل حالة التعيين $a, b, c = 'xyz'$ سوف يتم تعيين القيمة 'x' إلى المتغير a والقيمة 'y' إلى المتغير b والقيمة 'z' إلى المتغير c . هذه الآلية تناسب الدوال التي 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.

مثل المجموعات ، القوائم Lists هي سياق مرتب من القيم ، عندما تكون كل قيمة معرفة بفهرس . القاعدة اللغوية للتعبير عن عناصر القائمة يشبه تلك المستخدمة في المجموعات ؛ الفرق هو أننا نستخدم أقواسا مربعة بدلا من تلك الدائرية التي 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 :

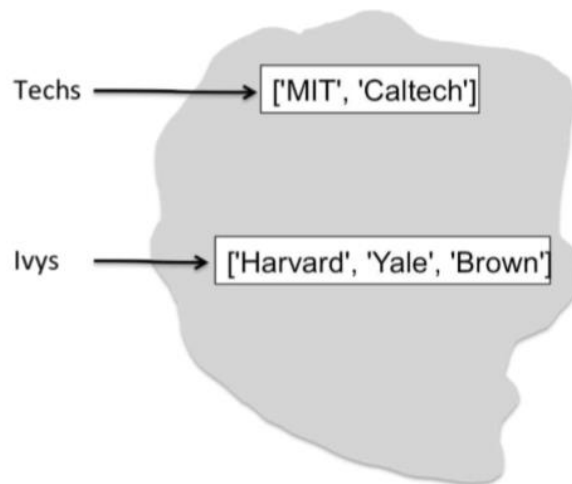
```
I did it all
4
love
```

بالمناسبة ، حقيقة استخدام الاقواس المربعة هي لأنه حرفيا نوع البيانات - القوائم Lists - مفهومة الى عناصر ، وتشريح القوائم ربما يؤدي 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 في الممارسة ، لانه في معظم الوقت يتم بناء القوائم عن طريق الاطراد وليس حرفيا كما يحدث في المجموعات ، وتختلف القوائم عن المجموعات في أن الأولى قابلة للتغير Mutable . في المقابل ، المجموعات والسلاسل الحرفية غير قابلة للتغير 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 في تلاوة التعويذة .

في 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 بإنشاء قائمتين ، وربط كل واحدة بالاسم المناسب لها كما فى الشكل التالى :



الشكل 5 - 1 قائمتين

ولنلاحظ أيضا فى عمليات التعيين الآتية :

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

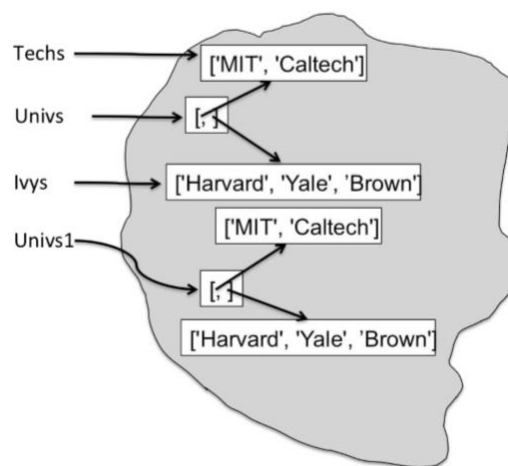
هنا أيضا يقوم المترجم بإنشاء قائمتين جديدتين وربط لهما المتغيرات كما يبدو لنا . وعناصر هذه القوائم هى نفسها قوائم ،
لنأخذ أوامر الطباعة الثلاثة الآتية :

```
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 `univ1` قد تم ربط كل منهما الى قيمة مغايرة تماما للآخرى :



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 `id` والتي 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 semantics تقول 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 والتي 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])
```

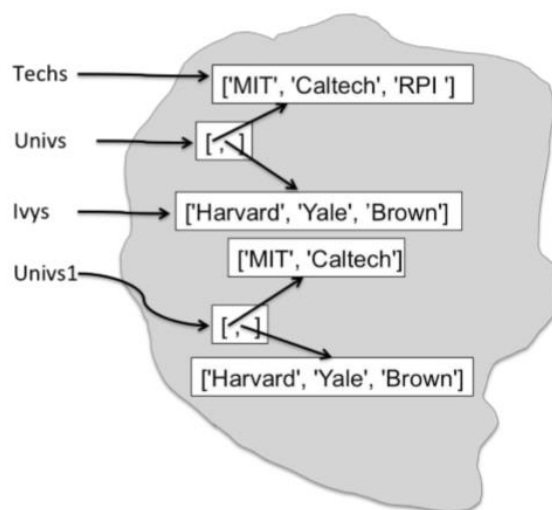
وستكون النتيجة :

```
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 تأثيري ، فبدلاً من إنشاء قائمة جديدة ، هي تقوم بالتعديل في القائمة الموجودة بالفعل `Techs` عن طريق إضافة عنصر جديد ، وسيتم إضافة النص 'RPI' بعد آخر العناصر الموجودة . وبعد الحاق العنصر ستكون حالي الحوسبة كالتالي :



الشكل 5 - 3 الاثبات لعملية التحور

القائمة `univ` لازالت تحتوي على نفس القائمتين ، ولكن محتوى هاتين القائمتين قد تغيرتا ، فلنقم بإجراء أمر الطباعة التالي :

```
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 شيء يسمى `Aliasing` - الاسم المستعار - ، يوجد هناك مساران متممايزان ، 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 مناسباً ، لكنه أيضاً قد يكون غادراً . تؤدي 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 تحول 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 Cloning النسخ

على الرغم من أنه متاح ، لكن تجنب التغيير 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 فى القائمة باستخدام عداد داخلي يتزايد فى نهاية كل عملية تكرار. عندما تصل قيمة العداد إلى الطول الحالي للقائمة تتوقف الحلقة ، هذا قد يؤدي كما قد نتوقع منه أن يفعل إذا لم يتم تحوّل القائمة داخل الحلقة التكرارية ، ولكن يمكن أن يكون لها عواقب مفاجئة إذا تم تحويل القائمة ، فى هذه الحالة ، يبدأ العداد المخفي عند 0 ، ويكتشف أن L1[0] فى L2 ، ويزيله - مما يقلل من طول L1 إلى 3. ثم يتم زيادة العداد إلى 1 ، وتستمر عملية التحقق مما إذا كانت قيمة L1[1] موجودة فى 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 `L1` بل 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 سياقاً جديداً 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 ومبهمه ، 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 ، نأخذ مثلا 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 تحتوي على دوال من النوع على الرتبة مثل map والتي تشبه هذه الدالة لكنها أكثر شمولية ، فلتعيين كل دالة الى كل عناصر القائمة L كما في الشكل 5-5 يمكننا استخدام الدالة map في أبسط اشكالها ، وهو ما يسمى الدالة الوحيدة (الدالة التي تحتوي على متغير وحيد) والمتغير الثانى عبارة عن أى ترتيب من المتغيرات التي يمكن ان تكون من أى نوع من القيم ، وتعيد لنا قائمة تم توليدها عن طريق تطبيق الدالة فى المتغير الاول الى كل عنصر من عناصر المتغير الثانى ، فمثلا ، التعبير map(factR, [1, 2, 6]) يعيد لنا القائمة [1, 2, 6] .

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

النصوص – المجموعات - القوائم Lists – Tuples – Strings

لقد رأينا ثلاثة أنواع 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

وأيضاً مقارنة لبعض الأشياء المتشابهة والمختلفة في كل نوع مختصرة كما في الشكل التالي :

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 التي تجعل المجموعات مفضلة هي كونها غير قابلة للتغير ، الاسماء البديلة ليست مقلقة ، وعدم قابليتها للتغير يجعل منها الحل الافضل لكي تستخدم كمفاتيح في قاموس كما سنرى في القسم التالي.

لأن النصوص strings تحتوى على أحرف فقط ، ربما هذا يجعلها أقل استخداما في البرمجة ، لكن على الجانب الآخر نجد العديد من الدوال الداخلية في اللغة والتي تجعل الحياة أكثر سهولة عند التعامل مع النصوص ، سيكون هذا موضحا باختصار في الجدول التالي :

وصف	الدالة
حسب كم مرة تكرر النص s1 بداخل النص s	s.count(s1)
يعيد رقم الفهرس لاول حرف من النص الفرعى s1 اذا كان موجودا داخل النص s أو تعيد قيمة -1 اذا لم يتم العثور على النص الفرعى	s.find(s1)
في الدالة السابقة ولكنها تعمل بشكل عكسى أى تبدأ من النهاية (الحرف r يشير الى (reverse	s.rfind(s1)

فس الدالة find ولكنها تعيد حالة اعتراض (انظر الفصل السابع) exception فى حالة عدم العثور على النص الفرعى	s.index(s1)
فس الدالة index ولكنها تعمل بشكل عكسى	s.rfind(s1)
بدل حالى كل الأحرف الى smallCase	s.lower()
قوم باستبدال كل النصوص الفرعية التى توافق القيمة old بالقيمة الجديدة new	s.replace(old, new)
قوم هذه الدالة بحذف المساحات البيضاء	s.rstrip()
ستطيع تقسيم النص على أساس محدد delimiter ويعيد قائمة تحتوى على نصوص المقطعة واذا غاب المتغير d يتم التقسيم على أساس محددات افتراضية ان (space, tab, newline, return, formFeed	s.split(d)

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 مجمدة {} 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 [1, 2, Mar, 'Feb', 5, 'Apr', 'Jan', 'May', '] . [3, 4

عند استخدام الحلقة التكرارية for فان القيم التى يتم اضافتها الى القائمة 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 بالشئ 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 هي كائنات mutable أى قابلة للتغيير ، فلا بد 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'` .

القوائم ، 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 :

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

الشكل 5-10 بعض الاساليب المرتبطة بالبيانات من النوع DICT

معظم لغات البرمجة لا تحتوى على مثل هذا النوع من البيانات ، لكن المبرمجين يبتكرون حيلاً 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 مفتاحاً ستعيد القيمة التى تقابل الفهرس 1 ، هذا قد يكون حلاً للمشكلة لكن تصور بدلاً 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 مطورو اللغة على ما يسمى بالتشفير hashing (سيتم شرحه فى الفصل العاشر) ، وهذا من شأنه تقليص وقت التنفيذ الى أقل ما يمكن .

6 الاختبار والتصحيح TESTING AND DEBUGGING

نحن نكره هذا الطرح ، ولكن ، قد كان الدكتور Dr. Pangloss على خطأ ، نحن لا نعيش في أفضل مكان في عالم الممكن ، هناك مناطق باردة على الأرض ، وهناك مناطق حارة ، هناك مناطق باردة أكثر من الطبيعي في الشتاء ، وأخرى أكثر حرارة في فصل الصيف ، ربما ينخفض سوق الأسهم - كثيرا - وربما الأسوأ ألا تعمل برامجنا بشكل صحيح من المرة الأولى ، تم اخراج هذا الفصل ليعلمنا كيف نتعامل مع هذه المشكلة ، وهناك الكثير أيضا في انتظارنا لتتعلمه بقراءة هذا الكتاب .

ومع ذلك ، وحرصًا على تزويدك ببعض التلميحات التي قد تساعدك في الحصول على هذه التلميحات ، المشكلة التالية قد جاءت في الوقت المناسب ، يقدم هذا الفصل مناقشة مكثفة للغاية حول الموضوع . في حين أن جميع الأمثلة البرمجية موجودة في بايثون ، فإن المبادئ العامة تنطبق على أي نظام معقد وتعيده للعمل .

الاختبار testing هو عملية تشغيل برنامج لمحاولة التحقق مما إذا كان يعمل أم لا. أمّا التصحيح debugging هو محاولة إصلاح البرنامج الذي تعرف أنه بالفعل لا يعمل على النحو المنشود .

لا يعتبر الاختبار وتصحيح الأخطاء عمليات يجب عليك البدء في التفكير فيها بعد إنشاء أحد البرامج ، يقوم المبرمجون الجيدون بتصميم برامجهم بطرق تجعل من السهل اختبارها وتصحيحها ، المفتاح لفعل هذا هو تقسيم البرنامج إلى مكونات منفصلة يمكن تنفيذها واختبارها وتصحيحها بشكل مستقل عن المكونات الأخرى .

حتى هذه المرحلة من الكتاب ، ناقشنا فقط آلية واحدة لدمج البرامج ، الدوال Functions . لذلك ، في الوقت الحالي ، ستستند جميع أمثلتنا حول الدوال ، و عندما نصل إلى آليات أخرى ، التصنيفات Classes بصفة خاصة ، سنعود إلى بعض المواضيع التي يتناولها هذا الفصل .

الخطوة الأولى في دفع برنامج للعمل هو جعله متوافق مع نظام ومتطلبات اللغة التي سوف يعمل على نظامها ، أي التخلص من أخطاء التركيب syntax والأخطاء الدلالية الثابتة semantics التي يمكن اكتشافها بدون تشغيل البرنامج. إذا لم تكن قد تجاوزت هذه النقطة في البرمجة ، فأنت لست مستعدًا لهذا الفصل ، اعمل قضاء بعض الوقت في العمل على برامج صغيرة ، ثم عد الى هنا .

6 – 1 اختبار Testing

أهم ما يمكن قوله عن الاختبار هو أن الغرض منه هو إظهار وجود أخطاء ، وليس لإظهار أن البرنامج خالي من الأخطاء. على حد تعبير إدجر دجكسترا ، "يمكن استخدام اختبار البرنامج لإظهار وجود أخطاء ، وليس لإثبات غيابهم أبدًا!" أو كما قال ألبرت أينشتاين ذات مرة ، "لا يمكن لأي قدر من التجارب إثبات الحقيقة ، لكن كل تجربة تثبت لي خطأ" .

لماذا هو كذلك؟ حتى أبسط البرامج يمكن أن يتولد لديه مليارات من المدخلات الممكنة ، خذ بعين الاعتبار ، على سبيل المثال ، البرنامج الذي يزعم تلبية المواصفات :

```
def isBigger(x, y):
    """Assumes x and y are ints
    Returns True if x is less than y and False otherwise"""
```

تشغيله على جميع أزواج الأعداد الصحيحة سيكون ، على أقل تقدير ، شئ ممل. أفضل ما يمكننا القيام به هو تشغيله على أزواج من الأعداد الصحيحة التي لديها احتمالية معقولة لإنتاج إجابة خاطئة إذا كان هناك خلل في البرنامج .

مفتاح الاختبار هو العثور على مجموعة من المدخلات ، تسمى مجموعة اختبار ، والتي لديها احتمالية عالية للكشف عن الأخطاء ، ولكن لا تستغرق وقتاً طويلاً للتشغيل ، و المفتاح لفعل هذا هو تقسيم كافة المدخلات الممكنة إلى مجموعات فرعية توفر معلومات متكافئة حول صحة البرنامج ، ثم إنشاء مجموعة اختبار تحتوي على إدخال واحد من كل قسم. (وفى العادة ، بناء مثل هذا الاختبار التجريبي ليس ممكناً في الواقع . ولكن فكر في هذا كمثال غير قابل للتحقيق) .

كل مجموعة من التقسيمات يتم تعيينها في مجموعة فرعية بحيث ينتمي كل عنصر في المجموعة الأصلية إلى واحدة من المجموعات الفرعية ، خذ بعين الاعتبار ، على سبيل المثال ، `isBigger(x, y)` . مجموعة من المدخلات الممكنة لهذه الدالة هي جميع مجموعات من الأعداد الصحيحة. تتمثل إحدى طرق تقسيم هذه المجموعة في المجموعات الفرعية السبعة التالية:

- x positive, y positive
- x negative, y negative
- x positive, y negative
- x negative, y positive
- x = 0, y = 0
- x = 0, y ≠ 0
- x ≠ 0, y = 0

إذا قمت باختبار التطبيق على قيمة واحدة على الأقل من كل مجموعة من هذه المجموعات الفرعية ، سيكون هناك احتمال معقول (ولكن ليس مضمون) لكشف الخطأ إذا وجد .

بالنسبة لمعظم البرامج ، فإن العثور على تقسيم جيد للمدخلات أسهل بكثير من فعله . عادة يعتمد الناس على الاستدلال على أساس استكشاف مسارات مختلفة من خلال مزيج من الكود والمتطلبات . الاستدلال على أساس استكشاف مسارات من خلال الكود تقع تحت ما يسمى اختبار صندوق زجاجي Glass-Box-Test . الاستدلال على أساس استكشاف المسارات من خلال المتطلبات تقع تحت ما يسمى اختبار الصندوق الأسود Black-Box-Test .

6-1-1 اختبار الصندوق الأسود Black Box Testing

من حيث المبدأ ، يتم إنشاء اختبارات الصندوق الأسود دون النظر إلى الكود المراد اختباره . يسمح اختبار الصندوق الأسود باختيار المخبّرين Testers والمنفّذين Implementers من مجموعات منفصلة . نحن نقوم عادةً بابتكار حالات اختبار كتلك ، أولئك الذين يقومون بتدريس دورات البرمجة يخلّقون حالات اختبار لمجموعة الطلاب لاختبارهم واعطائهم درجات عليها ، عندئذ فإننا نقوم بتطوير مجموعات اختبار الصندوق الأسود. غالبًا ما يكون لمطوري البرامج التجارية مجموعات منفصلة لضمان الجودة وتكون مستقلة إلى حد كبير عن مجموعات المطورين .

هذا الاستقلال يقلل من احتمالية توليد أخطاء مرتبطة بأخطاء في الكود. لنفترض ، على سبيل المثال ، أن كاتب أحد البرامج قدم افتراضًا ضمنيًا ، ولكنه غير صالح ، افترض أنه لن يتم استدعاء الدالة برقم سالب. إذا قام نفس الشخص ببناء حاشية الاختبار للبرنامج ، فمن المحتمل أن يكرر هذا الخطأ ، ولا يختبر الوظيفة بمتغيرات سالبة .

خاصية أخرى أيجابية في اختبارات الصندوق الأسود ، هي أنها قوية فيما يتعلق بالتغييرات في التنفيذ. نظرًا لأنه يتم إنشاء بيانات الاختبار دون معرفة بكيفية التنفيذ ، فلا يتم تغييرها عند تغيير التنفيذ .

كما قلنا سابقًا ، فغن أفضل الطرق لتوليد اختبار الصندوق الأسود هو استكشاف المسارات من المتطلبات ، فلنرى هذا المتطلبات :

```
def sqrt(x, epsilon):
    """Assumes x, epsilon floats
       x >= 0
       epsilon > 0
       Returns result such that
       x-epsilon <= result*result <= x+epsilon"""
```

يبدو أنه لا يوجد سوى مسارين مختلفين فقط من خلال هذه المتطلبات : الاول هو المتوافقة $x = 0$ والآخر المتوافقة $x > 0$. ومع ذلك ، فإن الحدس السليم يخبرنا ، في حين أنه من الضروري اختبار هاتين الحالتين ، فإن ذلك لا يكاد يكفي .

يجب أيضًا اختبار شروط الحدود* أيضا . عند النظر إلى القوائم ، غالبًا ما يعني ذلك النظر إلى القائمة الفارغة ، وقائمة تحتوي على عنصر واحد فقط ، وقائمة تحتوي على قوائم . عند التعامل مع الأرقام ، يعني ذلك عادةً النظر إلى القيم الصغيرة جدًا وكبيرة جدًا على أنها قيم متطابقة. بالنسبة لـ sqrt ، قد يكون من المنطقي محاولة اختبار قيم x و ϵ بطريقة مشابهة لتلك الموجودة في الجدول التالي :

x	epsilon
0.0	0.0001
25.0	0.0001
0.5	0.0001
2.0	0.0001
2.0	1.0/2.0**

	64.0
1.0/2.0**	1.0/2.0**
64.0	64.0
2.0**64.	1.0/2.0**
0	64.0
1.0/2.0**	2.0**64.
64.0	0
2.0**64.	2.0**64.
0	0

الصفوف الأربعة الأولى مخصصة لتمثيل الحالات النموذجية. لاحظ أن قيم x تتضمن مربعًا كاملاً ، ورقمًا أقل من رقم واحد ، ورقمًا به جذر تربيعي غير منطقي. إذا فشلت أي من هذه الاختبارات ، هناك خلل في البرنامج و يحتاج إلى إصلاح.

تختبر الصفوف المتبقية قيمًا كبيرة وصغيرة جدًا من x و ϵ . إذا فشل أي من هذه الاختبارات ، فلا بد أن هناك شيئًا ما يجب إصلاحه . شيء ما يحتاج الى اصلاح في التعليمات البرمجية أو تحتاج إلى تغيير المتطلبات حتى يكون من السهل تليتها . قد يكون ، على سبيل المثال ، من غير المعقول أن نتوقع قيم تقريبية لجذر مربع عندما يكون ϵ صغيرًا بشكل يبعث على السخرية .

* شرط الحدود : يعبر عن الشرط اللازم لتساوى الحدود في المعادلات الرياضية

وشرط حدود آخر لابد أن نفكر فيه وهو ما يتعلق بالأسماء المزيفة ، انظر الى الكود التالى :

```
def copy(L1, L2):
    """Assumes L1, L2 are lists
       Mutates L2 to be a copy of L1"""
    while len(L2) > 0: #remove all elements from L2
        L2.pop() #remove last element of L2
    for e in L1: #append L1's elements to initially empty L2
        L2.append(e)
```

سوف تظل تعمل هذه الدالة طوال الوقت بشكل سليم ، ولكن ليس عندما تشير L2 , L1 الى نفس القائمة . قائمة الاختبارات لا تتضمن الاختبار $\text{copy}(L, L)$ ، سوف لا يطر النتيجة المرغوبة مع انه لن يقود الى وجود خطأ بالكود .

6 - 2 - 2 اختبار الصندوق الزجاجي glass-Box Test

اختبار الصندوق الاسود لايمكن تخطيه ، لكنه ليس كافيا بالمره ، بدون النظر داخل الكود يصعب تحديد نوعيه الاختبارات اللازمة للتحقق من صحة البرنامج ، ففي المثال البسيط التالى :

```
def isPrime(x):
    """Assumes x is a nonnegative int
    Returns True if x is prime; False otherwise"""
    if x <= 2:
        return False
    for i in range(2, x):
        if x%i == 0:
            return False
    return True
```

بالنظر الى الكود يمكننا أن ندر أنه وبسبب الشرط $x \leq 2$ فإن القيم 0, 1, 2 يتم معاملتها على أنها حالات خاصة ، ولهذا السبب يجب اختبارها ، بدون النظر الى داخل الكود قد لا تفكر فى اختبار `isPrime(2)` ، لاننا عند اختبارها وكانت النتيجة `False` الأمر الذى يعنى وجود خطأ حيث يشير الى أن 2 ليست عدد أولى .

عادةً ما تكون مجموعات اختبار الصندوق الزجاجى أسهل في البناء من مجموعات اختبار الصندوق الأسود . و عادةً ما تكون المواصفات غير مكتملة وغالبًا ما تكون سيئة جدًا ، مما يجعلها تحديًا لتقدير مدى استفادة مجموعة اختبار الصندوق الأسود من مساحة المدخلات المثيرة للاهتمام .

وعلى النقيض من ذلك ، فإن فكرة استكشاف مسار من خلال الكود هى فكرة محددة بشكل جيد ، ومن السهل نسبياً تقييم مدى دقة استكشاف المساحة ، في الواقع ، الأدوات التجارية التي يمكن استخدامها لقياس استيفاء اختبارات صندوق زجاجي بشكل موضوعي .

مجموعة اختبار صندوق زجاجي تكون مكتملة المسار Path-Complete إذا كانت تختبر كل مسار محتمل من خلال البرنامج . وهو ما يستحيل تحقيقه ، لأنه يعتمد على عدد المرات التي يتم فيها تنفيذ كل حلقة وعمق كل عملية عودية recursion. على سبيل المثال ، التنفيذ العودي لدالة المضروب factorial function يتبع مسارًا مختلفًا لكل المدخلات الممكنة (لأن عدد مستويات التكرار سيختلف) .

علاوة على ذلك ، حتى جناح اختبار المسار المكتمل لا يضمن أن جميع الأخطاء سيتم كشفها . انظر :

```
def abs(x):
```

```

"""Assumes x is an int
Returns x if x >= 0 and -x otherwise"""
if x < -1:
    return -x
else:
    return x

```

تشير المتطلبات إلى أن هناك حالتين محتملتين ، x إما سالبة أو ليست كذلك. هذا يشير إلى أن مجموعة المدخلات $\{-2, 2\}$ قد تكون كافية لاستكشاف جميع المسارات في المتطلبات . يحتوي جناح الاختبار هذا على خاصية جميلة إضافية لإجبار البرنامج عبر جميع مساراته ، لذا يبدو كجناح صندوق زجاجي كامل كذلك . المشكلة الوحيدة هي أن هذا الجناح التجريبي لن يكشف حقيقة أن القيمة المطلقة (-1) ستعيد -1.

على الرغم من محدودية اختبار الصندوق الزجاجي ، فهناك عدد قليل من القواعد الأساسية التي عادة ما تكون جديرة باتباعها :

- اختبار كل الفروع للحالات الشرطية
- تأكد من أن كل استثناء يعطى النتيجة الصحيحة عند التنفيذ (انظر الفصل السابع)
- لكل حالة تكرارية من النوع for يوجد حالات اختبار عندما :
 - عندما لا تدخل الحلقة - كن متأكدت من تجربة الحلقة على القوائم الفارغة عندما يكون مطلوب منها الدخول -
 - عندما يتم تنفيذ حاشية الحلقة التكرارية مرة واحدة
 - عندما يتم تنفيذ حاشية الحلقة التكرارية مرات عديدة
- لكل حلقة تكرارية من النوع while يوجد أيضا حالات اختبار عندما :
 - نفس الحالات المتبعة مع الحلقات من النوع for
 - افحص جميع حالات الاختبار التي تطابق جميع الطرق الممكنة للخروج من الحلقة . على سبيل المثال ، عند بدء حلقة بـ $\text{len}(L) > 0$ when وليس $L[i] == e$ ، عليك العثور على الحالات التي تخرج فيها الحلقة عندما يكون $\text{len}(L)$ أكبر من الصفر والحالات التي تخرج فيها لأن $L[i] == e$
- فى حالات اعادة استدعاء الدوال function recursive call اختبر جميع القيم التي تجعل الدالة لا تتكرر ، أو تتكرر مرة واحدة ، أو تتكرر مرات عديدة .

6 - 1 - 3 اختبار التصرف Conducting Tests

غالباً ما يُعتقد أن الاختبار يتم على مرحلتين. ينبغي للمرء أن يبدأ دائماً باختبار الوحدات . خلال هذه المرحلة ، يقوم المختبر ببناء واختبار اختبارات مصممة للتأكد مما إذا كانت وحدات الكود - مفردة - (مثل ، الدالات semantics) تعمل بشكل صحيح. ويتبع ذلك اختبار التكامل ، الذي تم تصميمه للتأكد مما إذا كان البرنامج ككل يتصرف على النحو المنشود . يدور المختبرين حول هاتين المرحلتين ، فان الفشل ان وجد فى اختبار التكامل يؤدي ذلك الى اجراء تعديل فى الوحدات .

اختبار التكامل يكاد يكون دائماً أكثر تحدياً من اختبار الوحدة. وأحد أسباب ذلك هو أن السلوك المقصود للبرنامج بكامله غالباً ما يكون أصعب بكثير في تحديد خصائص السلوك المقصود لكل جزء من أجزائه. على سبيل المثال ، يعد توصيف السلوك المقصود لمعالج الكلمات أكثر تحدياً بشكل كبير من توصيف سلوك دالة تقوم بحساب عدد الحروف في الوثيقة. مشاكل الحجم يمكن أن تجعل اختبار التكامل صعباً أيضاً. انه لمن غير المعتاد أن تستغرق اختبارات التكامل ساعات أو حتى أيام للتنشغيل .

فى الغالب يكون لدى العديد من منظمات تطوير البرمجيات الصناعية مجموعات ، مجموعة لضمان جودة البرمجيات (SQA) منفصلة عن المجموعة المكلفة بتنفيذ البرنامج . تتمثل مهمة هذه المجموعة في التأكد من أنه قبل إصدار البرنامج يكون مناسباً للغرض المقصود . في بعض المؤسسات ، تكون مجموعة التطوير مسؤولة عن اختبار الوحدات وتكون مجموعة ضمان الجودة مسؤولة عن اختبار التكامل .

فى العمليات الصناعية ؛ أحيانا تتم عملية الاختبار بشكل آلى بالكامل ، لا يجلس المختبرون وراء المناضد يقذفون بالمدخلات وينتظرون المخرجات لفحصها ، وبدلاً من ذلك يستخدمون سواقات الاختبار والتي يتوقع منها أن :

- تشغيل البيئة اللازمة لانطلاق البرنامج (أو الوحدة) المراد اختبارها
- إطلاق البرنامج (أو الزحدة) المراد اختبارها وتزويده بقيم معطيات arguments سواء كانت مدخلة بشكل افتراضى أو يتم توليدها آلياً أثناء الانطلاق فى هيئة مدخلات متتالية .
- حفظ النتائج التى يحصل عليها
- التحقق من مطابقتها للنتائج المتوقعة
- اعداد التقرير اللذيذ

أثناء اجراء الاختبار نحتاج لبناء بعض الحواشى stub كتلك السواقات test drivers ، السواقات تحاكي اجزاء البرنامج التى تستخدم الوحدات التى يتم اختبارها ، بينما الحواشى تحاكي أجزاء البرنامج التى تستخدمها تلك الوحدات . الحواشى Stubs مفيدة وعملية ، لانها تمكننا من اختبار وحدات تحتاج الى برمجيات Softwar وفى بعض الاحيان حتى لو كانت تعتمد على أجهزة hardware غير موجود . وهذا الشئ رائع حقاً ، فهو يمكن مجموعات المبرمجين من التطوير المحاكى ، والاختبار لكثير من أجزاء النظام .

ويتوقع من تلك الحواشى أن :

- تتأكد من ملامة البيئة environment والمعطيات arguments للمتطلبات التى يضعها المبرمج فى البرنامج (استدعاء دالة بمعطيات غير ملائمة حتماً يؤدي الى أخطاء جسيمة)
- وان وُجِدَ أى اختلاف يستدعى التعديل على المعطيات والمتغيرات العامة Global variables فانها تقوم به حتى تتوافق مع المتطلبات
- ارجاع القيم حسب المتطلبات
- ارجاع القيم حسب المتطلبات

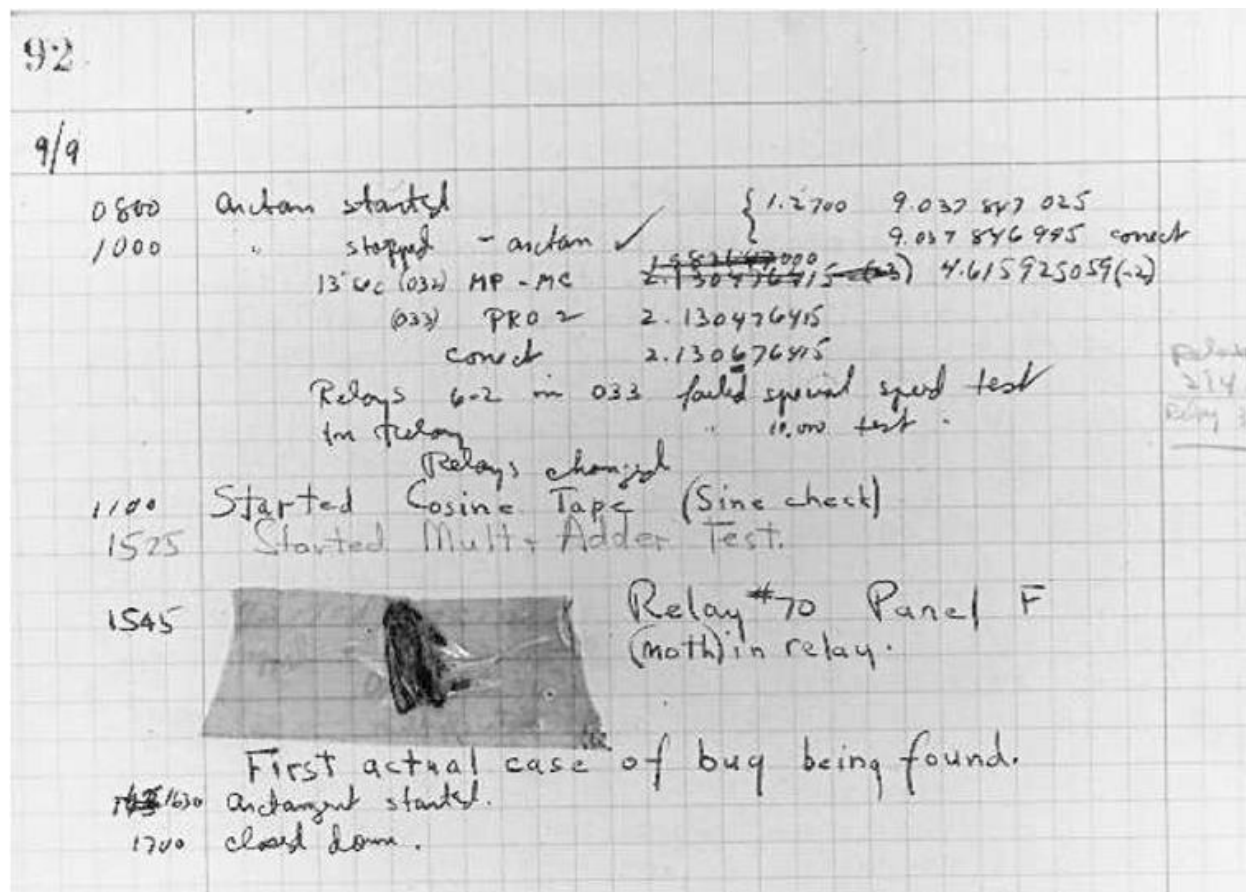
التحدى هنا هو أن تقوم ببناء حشوات بشكل كافٍ ، فاذا كانت الوحدة التى تستبدل بالحشوات تقوم بأداء مهام معقدة ، فان بناء حشوات تقود بأداء ما يتفق مع المتطلبات يكون بمثابة كتابة البرنامج الذى من المفترض أن يستبدل بعض أجزائه بالحشوات .

هناك طريقة واحدة للتغلب على هذه المشكلة ، هو تقليل المدخلات المقبولة عن طريق الحشوات ، وإنشاء جودل يحتوى على المخرجات المتوقعة والتي تقابل كل مدخل لاستخدامه فى مجموعة الاختبار .

الشئ الجذاب فى ان عملية الاختبار تتم آلياً ، هو أنه يسهل اختبار الانحدارية regression testing . عندما يحاول المبرمجون عمل تصحيح debug لبرنامج ما ، فمن الشائع جداً تثبيت "إصلاح" Fix يكسر شيئاً كان معتاداً أن يعمل ، و عند إجراء أي تغيير ، مهما كان صغيراً ، يجب أن نتحقق من أن البرنامج لا يزال يمر بجميع الاختبارات التى كان يمر بها .

6 - 2 التقية Debugging

هناك أسطورة ساخرة حول كيفية التعرف على عملية إصلاح العيوب في البرنامج باسم Debugging . الصورة أدناه في 9 سبتمبر 1947 ، عبارة عن صفحة في كتاب مختبر من المجموعة التي تعمل على حاسبة "مارك 2" المتناوبة في جامعة هارفارد



وقد ادعى البعض أن اكتشاف حشرة البق التي وقعت في أجزاء حاسبة مارك 2 أدى إلى استخدام عبارة تصحيح الأخطاء Debugging . على أية حال ، فإن الصياغة ، "First actual case of a bug being found" ، تشير إلى أن التفسير الحرفي للكلمة كان شائعاً بالفعل .

أوضحت غرايس موراي هوبر ، القائدة في مشروع حاسبة مارك 2 ، أن مصطلح "bug" كان بالفعل يستخدم على نطاق واسع لوصف مشاكل الأنظمة الإلكترونية خلال الحرب العالمية الثانية .

وقبل ذلك بفترة وجيزة ، شمل التعليم الحديث للكهرباء في هوكينز ، وهو كتيب كهربائي عام 1896 ، على المقتبس "bug" is "used to a limited extent to designate any fault or trouble in the connections or working of electric apparatus". يُستخدم مصطلح "bug" بشكل محدد لتحديد أي خطأ أو مشكلة في الاتصالات أو عمل في أجزاء الأجهزة الكهربائية. في اللغة الإنجليزية ، كلمة "bugbear" تعني "أي شيء غير ضروري أو بدون داع أو مفرط

يبدو أن شكسبير قد اختصر هذا إلى "bug" ، عندما كتب عن تدمير هاملت "البق والعفاريت في حياتي".

إن استخدام كلمة "bug" يؤدي أحياناً إلى تجاهل الناس للحقيقة الأساسية المتمثلة في أنه إذا كتبت برنامجاً ووجدت "خللاً به" ، فقد أفسدت الأمر . البق لا يزحف بدون داع داخل برنامج خال من البق . إذا كان برنامجك به خطأ ، فذلك لأنك وضعته هناك . البق لا تتكاثر في البرامج. إذا كان برنامجك يحتوي على أخطاء متعددة ، فذلك لأنك ارتكبت عدة أخطاء .

يمكن تصنيف أخطاء وقت التشغيل Runtime Bugs دائماً على أساس بعدين :

خفى : Overt >--- علنى : covert

يحتوي الخطأ العلنى Overt Bug على مظهر واضح ، على سبيل المثال ، تعطل البرنامج أو أنه يستغرق وقتًا أطول (ربما إلى الأبد) للتشغيل مما ينبغي.

الخطأ الخفى Covert Bug ليس له مظهر واضح. قد يتم تشغيل البرنامج حتى لا ينتهي بأي مشكلة - بخلاف تقديم إجابة غير صحيحة .

تقع العديد من الأخطاء بين خفى وعلنى ، وما إذا كان الخطأ علنا أم لا ، يمكن أن يعتمد على مدى دقة فحص سلوك البرنامج .

دائم : Persistent → متقطع intermittent

يحدث خطأ دائم في كل مرة يتم فيها تشغيل البرنامج بنفس المدخلات . بينما يحدث الخطأ المتقطع لبعض الوقت فقط ، حتى عندما يتم تشغيل البرنامج على نفس المدخلات وعلى ما يبدو تحت نفس الظروف . عندما نصل إلى الفصل 12 ، سنبدأ في كتابة برامج من النوع الذي تكون فيه الأخطاء المتقطعة شائعة .

أفضل أنواع الأخطاء (bugs) أن تكون علنية ومستمرة. ينبغي ألا يندفع المطورين بسهولة نشر البرامج . فحتى لو قام شخص أحق باستخدام تلك البرامج ، فسوف يكتشف بسرعة حماقاتهم . ربما يقوم البرنامج بشيء فظيع قبل توقعه عن العمل ، على سبيل المثال ، حذف الملفات ، ولكن على الأقل سيكون لدى المستخدم سبب للقلق (إن لم يكن الذعر).

يحاول المبرمجون الجيدون أن يكتبوا برامجهم بطريقة تجعل أخطاء البرمجة تؤدي إلى أخطاء تكون علنية وثابتة. هذا ما يسمى في كثير من الأحيان البرمجة الدفاعية defensive programming .

هناك نقطة أخرى لا يمكن قبولها ، إذا كانت الأخطاء (Bugs) علنية ولكنها متقطعة . سيكون ذلك خطيرا عندما يتواجد فى أنظمة مراقبة المرور الجوى لرصد أماكن الممرات وإرشاد الطائرات الى الطريق الصحيح ، حتما سيؤدي ذلك الى كوارث فادحة ، ستشعر حينها أنك تسكن فى جنة مليئة بالحمقى .يستطيع المرء أن يعيش فى الجحيم لبعض الوقت لكن لا يستطيع أن يقبع أمام برنامج يعج بالبقات ..

عاجلا أم آجلا ستظهر تلك البقات ، فإذا كانت الظروف التى تتطلب ظهور البقات يسهل إعادة انتاجها ، سيكون من السهل تحديدها وحصرها واصلاحها . أما اذا كانت تلك الظروف غير واضحة ، سيصبح التعايش معها فى غاية الصعوبة .

البرامج التى تقع فى منطقة البقات الخفية غالبا ما تكون خطيرة للغاية. وبما أنها لا تبدو منها أى مشاكل ، فإن الناس يستخدمونها ويثقون بها لفعل الشيء الصحيح . يعتمد المجتمع بشكل متزايد على برمجيات لأداء عمليات حسابية حاسمة تتجاوز قدرة البشر على القيام بها ، أو حتى التحقق من صحتها . لذلك ، يمكن للبرامج أن تقدم إجابة خاطئة لا يتم كشفها لفترات طويلة من الزمن. مثل هذه البرامج يمكن أن تسبب الكثير من الضرر .

يمكن للبرنامج يقوم بتقييم محفظة السندات والرهونات العقارية ، ويقدم تقييمات خاطئة . الامر الذى يمكن أن يكلف البنك (والمجتمع أيضا) الوقوع فى مخاطر جسيمة . إن آلة العلاج الإشعاعي التى تولد إشعاعات أكثر بقليل أو أقل بقليل مما هو مطلوب يمكن أن يكون الفرق بين الحياة والموت بالنسبة لشخص مصاب بالسرطان .

البرامج التى تحتوى على أخطاء خفية وتظهر بشكل مستمر أحيانا (وأحيانا لا) تسبب ضررا أقل من تلك التى تسبب تلك الأخطاء الخفية المتقطعة . وخلاصة القول أن أصعب أنواع الأخطاء فى الاصلاح هى تلك التى تكود خفية ومتقطعة الحدوث .

6 - 2 - 1 تعلم كيفية الفحص Learning To Debugging

التصحيح هو مهارة مكتسبة. لا أحد يفعل ذلك جيداً بشكل غريزي . الجميل أنه ليس من الصعب تعلمها ، وهي أيضا مهارة قابلة للتحويل . يمكن استخدام نفس المهارات المستخدمة في تصحيح البرامج لمعرفة ما هو الخطأ فى الأنظمة المعقدة الأخرى ، على سبيل المثال ، التجارب المعملية أو البشر المرضى .

منذ أربعة عقود على الأقل يقوم الناس ببناء برمجيات لغرض الفحص debugging وهناك أدوات مدمجة لفحص وتصحيح الأخطاء نندجة بداخل الـ IDLE . هذه البرمجيات يفترض لها أن تساعد البشر في البحث عن الأخطاء وتصحيحها . انها تستطيع المساعدة ، ولكن قليلا جداً . الأمر الأكثر أهمية هو كيفية التعامل مع المشكلة ، معظم المبرمجين الخبيرين لا يتعاملون مع أدوات الاختبار Debugging Tools . فهم يعتبرون أن أفضل أدوات البرمجة على الإطلاق هي حالة الطباعة Print .

يبدأ التصحيح عندما يثبت الاختبار أن البرنامج يتصرف بطرق غير مرغوب فيها . التصحيح هو عملية البحث عن تفسير لهذا السلوك . إن المفتاح لأن تكون دائماً جيداً في عملية تصحيح الأخطاء هي أن يتم بشكل منهجي في إجراء هذا البحث .

ابدأ بدراسة البيانات المتاحة . وهذا يشمل نتائج الاختبار ونص البرنامج. ادرس جميع نتائج الاختبار. لا تدرس فقط الاختبارات التي كشفت عن وجود مشكلة ، ولكن أيضاً تلك التي يبدو أنها تعمل بشكل مثالي. محاولة لفهم سبب نجاح اختبار أو وآخر لا . عند النظر إلى نص البرنامج ، ضع في اعتبارك أنك لا تفهمه تمامًا. إذا فعلت ، فربما لن يكون هناك خطأ

بعد ذلك ، قم بتشكيل فرضية تعتقد أنها متسقة مع جميع البيانات. يمكن أن تكون الفرضية ضيقة مثل "إذا قمت بتغيير السطر رقم 403 من $x < y$ إلى $x \leq y$ ، فإن المشكلة ستختفي" أو بشكل واسع مثل "لا يتم إنهاء البرنامج لأن لدي حالة خروج خاطئة في بعض الحلقات" .

بعد ذلك قم ببناء تجربة قابلة للتكرار مع امكانية دحض فرضية اتساق البيانات ، على سبيل المثال ، قد تقدم على كتابة حالة طباعة قبل وبعد كل حلقة تكرارية (من النوع while مثلا) ، اذا حصلت على نتيجة مزدوجة ، فان فرضية أن هذه الحلقة التكرارية تتسبب في تعطيل البرنامج هي فرضية خاطئة ، يتم دحضها على الفور . تذكر قبل أن تقوم ببناء التجربة ان تبني بعض التوقعات للنواتج المطلوبة فعلا (وليس التي تحصل عليها بعد الاختبار) . اذا حصلت على واحدة وانتظرت ظهور الاخرى فعليك أن تسجد شكرا لانك حصلت على موضع العطب .

أخيراً ، من المهم الاحتفاظ بسجل للتجارب التي جربتها . عندما تقضي عدة ساعات في تغيير شفرتك في محاولة لتعقب خللٍ بعيد المنال ، من السهل نسيان ما حاولت القيام به بالفعل . إذا لم تكن حريصًا ، فمن السهل إهدار عدد كبير جدًا من ساعات في تجربة التجربة نفسها (أو على الأرجح تجربة تبدو مختلفة ولكنها ستمنحك نفس المعلومات) مرارًا وتكرارًا. تذكر ، كما قال الكثيرون ، "الجنون أن تفعل الشيء نفسه ، مرارا وتكرارا ، ولكن تتوقع نتائج مختلفة."

6 - 2 - 2 تصميم التجربة Designing Expiement

اعتبر عملية التصحيح Debugging كعملية بحث ، وكل تجربة هي محاولة لتقليل حجم مساحة البحث. تتمثل إحدى طرق تقليل حجم مساحة البحث في تصميم تجربة يمكن استخدامها لتحديد ما إذا كانت منطقة معينة من الكود مسؤولة عن مشكلة تم اكتشافها أثناء اختبار التكامل (اختبار من الداخل الى الخارج). هناك طريقة أخرى لتقليل مساحة البحث وهي تقليل كمية بيانات الاختبار المطلوبة للامساك بأحد مناطق الخلل .

دعنا نلقي نظرة على مثال به خطأ مفتعل لنرى كيف يمكن للمرء أن يذهب نحو تصحيح الأخطاء. تخيل أنك كتبت رمز التحقق المتناظر في الشكل 6.1 وأنت واثق من مهاراتك في البرمجة التي وضعتها على الويب دون اختبارها. لنفترض أيضاً أنك تتلقى رسالة إلكترونية تقول: "لقد اختبرت البرنامج !**! الخاص بك على النص 1000 ، وطبعت نعم . أي أحقق بعد يمكن أن يرى أنه ليس متناظر . أصلحه من فضلك .

```
def isPal(x):
```

```
    """Assumes x is a list
```

```
    Returns True if the list is a palindrome; False otherwise"""
```

```
    temp = x
```

```
    temp.reverse
```

```

    if temp == x:
        return True
    else:
        return False

def silly(n):
    """Assumes n is an int > 0
    Gets n inputs from user
    Prints 'Yes' if the sequence of inputs forms a palindrome;
    'No' otherwise"""
    for i in range(n):
        result = []
        elem = raw_input('Enter element: ')
        result.append(elem)
    if isPal(result):
        print 'Yes'
    else:
        print 'No'

```

الشكل 6 – 1 برنامج مع Bugs

نستطيع تجربة المدخل 1000 ولكن لنجرب الان ادخال قيم بسيطة ، ستكون الدالة اكثر حساسية لهذا

```

Silly(2)
'Enter element: a
'Enter element: b

```

لحسن الحظ أنها أيضا تفشل في هذه الاختبارات البسيطة أيضا ، وهذا معناه أنك لا تحتاج لتجربة نص آخر ك 1000 مثلا ولكن أيضا لسوء الحظ انت لا تعرف لماذا . في حالتنا هذه ومع برنامج صغير الحجم كهذا ربما تكون قادرا على استخراج الخطأ بمجرد

النظر ، لكن دعنا نفترض أنه كبير بما يكفي ، وتعالوا معنا لكي نقوم ببناء تجربة التصحيح خطوة بخطوة والعمل على لتقليل حجم البحث في كل مرة كما تعلمنا .

7 - الفصل السابع الاستثناء والتوكيد Exception And Assertion

عادة ما يتم تعريف الاستثناءات Exceptions على أنها " شئ ما لا يوافق الطبيعة " . ومن ثم فهي تعتبر نادرة ، ولكن لا يوجد ما يشير الى ندرة ال Exceptions ليست نادرة فى بايثون ، انها موجودة فى كل مكان ، فى الواقع ، فإن كل وحدة متكاملة -Module- فى مكتبة بايثون القياسية تستخدمهم ، البايثون نفسها ، سوف تشير الى Exceptions وترفعه عندما توجد ملابسات تستدعى ذلك ، انت بالفعل قد رأيت بعضا منهم .

افتح برنامج Python shell واكتب :

```
test = [1,2,3]
test[3]
```

وسوف يرد عليك ال Interpreter بالرد التالى :

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    test[3] IndexError: list index out of rang
```

ال IndexError هو نوع ال Exception الذى أخرجه لنا البايثون عندما حاول البرنامج الوصول الى عنصر ليس موجود ضمن كائن مرتبة عناصره فى فهرس . النص الذى يلى ال IndexError يقدم لنا معلومات اضافية عن سبب ظهور هذا ال Exception .

معظم ال Built-in Exceptions تتعامل مع ال Programs التى تحاول تنفيذ Statements بها semantics غير مناسبة ، -سوف نتعامل فى اخر هذا الفصل مع استثناء الاستثناء Exceptional Exceptions التى لا تتعامل مع الأخطاء - هؤلاء القراء - اقصد أنتم - الذين فكروا فى كتابة وتنفيذ برامج باستخدام بايثون بالفعل سيقابلون العديد منها ، ومن بين هذه الاستثناءات الاكثر شيوعا .. خطأ القيمة ValueError أو خطأ بالنوع TypeError, أو خطأ بالتسمية NameError

7 - 1 الإمساك بالـ Exceptions

حتى الآن ، نحن لازلنا نعامل ال Exceptions على أنها Fatal خطيرة وقاتلة ، عندما يظهر Exception يتوقف البرنامج عن متابعة التنفيذ (ربما تكون كلمة ينهار مناسبة فى هذه الحالة) ونذهب الى الكود ونتابع فحص لنرى ما الخطأ الذى حدث ، فعندما يقفز أمام اعيننا ال Exception الذى تسبب فى انهيار البرنامج فنحن نطلق على هذا ظهور لـ Unhandled Exception او استثناء لم يتم الامساك به ..

لا بد ألا يتسبب ظهور الـ Exception فى انهيار البرنامج أو توقفه عن العمل ، الاستثناءات Exceptions يمكن – بل لا بد – أن يتم الامساك بها بواسطة البرنامج . أحيانا يظهر الاستثناء لان هناك خطأ Bug فى البرنامج (كمحاولة الوصول الى متغير ليس موجودا) لكن ، وفى أغلب الأحيان ، الاستثناءات يمكن أن تكون – ولا بد ان تكون – شيئا ما يمكن توقعه من قبل المبرمج ، فربما يحاول البرنامج الوصول الى ملف ليس موجودا ، أو أن برنامج تفاعلى يطلب من المستخدم ان يقوم بادخال معين ، لكن يقوم المستخدم بادخال غير ملائم .

وإذا كنت تعلم أنه فى سطر ما من سطور الكود خاصتك يمكن أن يقوم باظهار استثناء عند التنفيذ ، فعليك الامساك به ، فى البرامج المكتوبة جيدا ، فان عدم الامساك بالاستثناء هو الاستثناء بعينه .

تأمل هذا الكود :

```
successFailureRatio = numSuccesses/float(numFailures)
print 'The success/failure ratio is', successFailureRatio
print 'Now here'
```

أغلب الوقت سوف يعمل هذا البرنامج على أكمل وجه ، ولكمه سيفشل اذا تم ادخال قيمة لـ numFailures تساوى الصفر ، القسمة على الصفر سوف تتسبب جعل نظام تشغيل Python باظهار استثناء ZeroDivisionError ولذلك لن تظهر أبدا حالة الطباعة .

سوف يكون من الافضل كتابة الكود بالشكل التالى بحيث نتمكن من الامساك بالاستثناء حين ظهوره :

```
try:    successFailureRatio = numSuccesses/float(numFailures)
        print 'The success/failure ratio is', successFailureRatio
except ZeroDivisionError:
        print 'No failures so the success/failure ratio is undefined.'
        print 'Now here'
```

عندما يدخل البرنامج الى النطاق try فانه سوف يقوم بتقييم العملية numSuccesses/float(numFailures) فإذا كانت العملية ناجحة يقوم بتعيين المتغير successFailureRatio الى القيمة الناتجة ثم يتابع الى حالة الطباعة – أو أى حالات اخرى داخل النطاق ان وجدت – ثم يتخطى النطاق except الى الحالة التى تليه – نعى حالة الطباعة "Now Here" ، وإذا – أو اينما – ظهر استثناء من النوع ZeroDivisionError أثناء تقييم الناتج فسوف يقفز المؤشر على الفور الى الجزء except (ويتجاهل تعيين المتغير successFailureRatio الى القيمة الناتجة وحالات الطباعة) ، ويقوم بطباعة الجملة 'No failures so the success/failure ratio is undefined.' ثم يكمل الى حالة الطباعة خارج نطاق الـ try-except .

درب أصابعك ..

اكتب دالة عند تنفيذها تستوفى المتطلبات التالية :

```
def sumDigits(s):  
    """Assumes s is a string  
    Returns the sum of the decimal digits in s  
    For example, if s is 'a2b3c' it returns 5"""
```

لنتضح الامور أكثر ، تعالوا بنا نلق نظرة على الكود التالى :

```
val = int(raw_input('Enter an integer: '))
print 'The square of the number you entered is', val**2
```

سوف يقوم نظام تشغيل بايثون باستقبال القيم المدخلة له وتحويلها الى اعداد صحيحة ، وتربيعها وطباعة قيمة التربيع ، لكن ماذا لو قام المستخدم بادخال شيئا ليس بالاعداد الصحيحة او حتى الكسرية ؟، ماذا لو قام بادخال النص abc على سبيل المثال ؟ ، سيقوم نظام تشغيل بايثون باظهار الـ ValueError exception ولن يتم الوصول ابدا الى حالة الطباعة التالية .
ما يجب أن يكتبه المبرمج لابد أن يكون شيئا مماثلا لهذا :

```
while True:
    val = raw_input('Enter an integer: ')
    try:
        val = int(val)
        print 'The square of the number you entered is', val**2
        break #to exit the while loop
    except ValueError:
        print val, 'is not an integer'
```

بعد الدخول فى الحلقة التكرارية ، سيطالب البرنامج من المستخدم ادخال رقم صحيح ، وعندما يدخل المستخدم شيئا سيقوم البرنامج بالدخول فى النطاق try-except اذا لم تسفر الحالة الاولى عن استثناء خطأ فى القيمة ValueError ، سيتم طباعة القيمة وتنفيذا الحالة break للخروج من الحلقة ، ولكن اذا أسفر النطاق try عن استثناء ValueError ، سيتم نق المؤشر فورا الى النطاق except ، ولذلك ، اذا قام المستخدم بادخال قيمة ليست رقما ، سيخبر البرنامج المستخدم أن القيمة المدخلة ليست رقما ، ويطلب منه اعادة الادخال مرة أخرى ، بدون الخوف من ان يتسبب اى ادخال خاطئ من المستخدم فى اظهار استثناءات غير ممسك بها unhandled exceptions .

الجانب السلبي فى هذا الموضوع ان حجم البرنامج قد تضاعف من سطرين الى ثمانية سطور ، ولو ان هناك عدة ادخالات فى أماكن متفرقة من البرنامج سيكون الامر مرهقا بعض الشيء . بالطبع لن يكون هناك مشكلة اذا قمنا بانشاء function تتوقى عملية قراءة هذا الادخال .

```
def readInt():
    while True:
        val = raw_input('Enter an integer: ')
        try:
            val = int(val)
            return val
        except ValueError:
            print val, 'is not an integer'
```

والأفضل أن يتم تعميم هذه الدالة ليتم استخدامها في أغراض متعددة في كل مرة يطلب فيها ادخال من المستخدم :

```
def readVal(valType, requestMsg, errorMsg):
    while True:
        val = raw_input(requestMsg + ' ')
        try:
            val = valType(val)
            return val
        except ValueError:
            print val, errorMsg
```

الدالة (readVal) هي دالة متعددة الأوجه Polymorphic ، على سبيل المثال ، هي تعمل على مدخلات لجميع انواع المدخلات ، هذه الدوال هي سهلة الكتابة في بايثون طالما ان البيانات هي أنواع أولية first class values . نستطيع ان نسأل عن ادخال رقم صحيح كالتالى :

```
val = readVal(int, 'Enter an integer:', 'is not an integer')
```

ربما تظهر الاستثناءات على أنها غير صديقة – لأنها ، ان لم يتم الامساك بها ستتسبب فى توقف البرنامج – لكن تذكر الوجه الآخر ، مالىذ ينبغي أن يفعله تحويل النوع `int` ، فمثلا ، عندما يطلب منك تحويل النص "abc" الى كائن من النوع `int` ، ربما قد تفكر فى أن الدالة يمكنها أن تعيد رقم صحيح يعبر عن عدد ال bits اللازمة لتشفير الحرف ، لكن للأسف لم يكن المبرمج ينوى هذا ، وربما بدلا من ذلك قد تقول انها يمكن ان تعيد القيمة `None` ، وإذا لم تفعل فعلى المبرمج أن يضيف سطرًا ليتأكد مما اذا كانت العملية تعيد القيمة `None` أم لا ، والمبرمج الذى ينسى ذلك الإجراء ربما يعرض برنامجه للخطر ، لانه سوف يقوم بتقديم مخرجات غير مألوفة عند التشغيل ،

مالجديد اذا ، فما زال بتوجب على المبرمج اضافة سطر برمجى للتعامل مع هذا الاستثناء ، على كل حال اذا نسى المبرمج ادراج مثل هذا السطر سوف يتعطل البرنامج فورا ، ويظهر نظام تشغيل بايثون استثناء ، وهذا شىء جيد . ويقوم بارسال تحذير لمستخدم البرنامج بأن هناك شىء ما خطأ قد حدث ، (وكما ناقشنا فى الفصل السابق أن الاخطاء الظاهرة Overt أفضل بكثير من تلك التى تكون مخفية Covert) ، وعلاوة على ذلك فهى تزود المبرمج عندما يقوم بعمل تصحيح (Debugging) بأن هناك شيئا خطأ أو انحراف فى البرنامج .

اذا كان من الممكن لنطاق ما أن يرسل أكثر من نوع من الاستثناءات فبأمكان المبرمج ادراجهم فى مجموعة (Tuple) يتم وضعها بعد ال `except keyword` بهذا الشكل :

```
except (ValueError, TypeError):
```

سوف يتم الدخول الى النطاق `except` فى حالة ظهور اى استثناء من داخل النطاق `try` من الاستثناءات التى تم ادراجهم فى المجموعة ، بدلا من ذلك ، يمكننا كتابة حالة استثناء منفصلة لكل نوع على حدة ، لنتيح للبرنامج أن يختار من بينها بناءا على الاستثناء الذى يظهر ، فاذا كتب المبرمج الكلمة

```
except:
```

فإنه سوف يتم الدخول الى هذا النطاق اذا تسبب أى من الحالات الموجودة فى النطاق `try` باظهار استثناء ، هذا هو جل ما نرمى اليه من هذا القسم 7 – 1 .

7 – 2 الاستثناءات كآلية للسيطرة على التدفق

لا تفكر فى الاستثناءات أنها وجدت فى المقام الاول لتبيان الأخطاء ، لكن كآلية مناسبة جدا للتحكم فى التدفق يمكن استخدامها بشكل مبسط فى البرامج .

فى العديد من لغات البرمجة يكون المنهج الاساسى للتعامل مع الاخطاء هو بناء وظائف " functions " تقوم بارجاع قيم معينة (شىء ما مماثل للقيمة `None` فى بايثون) للإشارة الى أن هناك شيئا ما على خطأ ، كل تنفيذ لوظيفة `function` هو للتأكد من أن هذه القيمة تم ارجاعها ، فى بايثون ، من المعتاد جدا أن ترى وظيفة `function` تعلقى استثناء `Raise an Exception` عندما لا تكون قادرة على ارجاع قيمة حسب المتطلبات .

فى بايثون ، فان الحالة `raise` تقوم باطلاق استثناء مخصص لما يحدث وتكون الصيغة العامة لها :

```
raise exceptionName(argument)
```

ال Exception Name سيكون فى الغالب أحد اسماء الاستثناءات الملحقة باللغة ، وعلى كل حال يمكن للمبرمجين اضافة أسماء جديدة ، ويتم ذلك عن طريق بناء تصنيف فرعى subClass (كما سيرد فى الفصل 8) من التصنيف الاساسى الملحق باللغة Exception . الانواع المختلفة من الاستثناءات لها أنواع مختلفة من المعطيات (arguments) ، لكن فى أغلب الوقت ستكون المعطيات فى شكل نص مفرد يستخدم للتعبير عن سبب انطلاق الاستثناء .

درب أصابعك :

نفذ وظيفة function تفى بالمتطلبات الاتية :-

```
def findAnEven(l):  
    """Assumes l is a list of integers  
    Returns the first even number in l  
    Raises ValueError if l does not contain an even number"""
```

فكر فى المسألة التى عرضنا لها فى الشكل 7 – 1 :

```
def getRatios(vect1, vect2):
    """Assumes: vect1 and vect2 are lists of equal length of numbers
    Returns: a list containing the meaningful values of
    vect1[i]/vect2[i]"""
    ratios = []
    for index in range(len(vect1)):
        try:
            ratios.append(vect1[index]/float(vect2[index]))
        except ZeroDivisionError:
            ratios.append(float('nan')) #nan = Not a Number
        except:
            raise ValueError('getRatios called with bad arguments')
    return ratios
```

الشكل 7 – 2 – 1 استخدام Exception للتكم فى التدفق

الكود يظهر فى النطاق try حالتين استثناءيتين تستدعيان اطلاق نوعين من الـ Exceptions وجدت ، الاولى عندما يوجد من بين عناصر القائمة الثانية عنصرا قيمته تساوى الصفر مما سيتسبب فى اطلاق استثناء مشهور هو ZeroDivisionError والذى يظهر نتيجة وجود صفر فى الجانب الايمن من عملية القسمة والذى سوف تكون نتيجته nan وهى تمثل رقم ولكنه غير معروف (تخيل القسمة 1 على 0.5 تساوى 2 وإذا قسمنا 1 على 0.01 تساوى 100 فكلما كانت قيمة الرقم على الجانب الايمن فى عملية القسمة صغيرا كلما تضخمت قيمة الناتج .. كلما كانت قيمة الرقم صغيرة كلما اقتربت أكثر من الصفر فالقسمة على الصفر تعطر الرقم الذى يعتبر فرضا نهاية الأعداد التى هى أصلا لانهاية لها) ، وإذا انطلق أى استثناء آخر مخالف لاستثناء القسمة على صفر كأن يكون عدد العناصر فى القائمتين مثلا غير متساو مثلا

الكود التالى يصور كيف لبرنامج أن يستخدم الوظيفة (getRatios) ، الاسم msg فى النطاق except ValueError ويشير الى النص المدخل كمعطيات للخطأ ValueError عندما ينطلق .

عند التنفيذ :


```
try:
    print getRatios([1.0,2.0,7.0,6.0], [1.0,2.0,0.0,3.0])
    print getRatios([], [])
    print getRatios([1.0, 2.0], [3.0])
except ValueError, msg:
    print msg
```

سيكون الناتج :

```
[1.0, 1.0, nan, 2.0]
[]
getRatios called with bad arguments
```

الشكل التالي هو عملية تنفيذ لنفس الدالة بدون الاستخدام الآلية (حاول - اقبض على استثناء) try-except المستخدمة في الأمثلة السابقة :

```

def getRatios(vect1, vect2):
    """Assumes: vect1 and vect2 are lists of equal length of numbers
    Returns: a list containing the meaningful values of
    vect1[i]/vect2[i]"""
    ratios = []
    if len(vect1) != len(vect2):
        raise ValueError('getRatios called with bad arguments')
    for index in range(len(vect1)):
        vect1Elem = vect1[index]
        vect2Elem = vect2[index]
        if (type(vect1Elem) not in (int, float)) or (type(vect2Elem) not in (int, float)):
            raise ValueError('getRatios called with bad arguments')
        if vect2Elem == 0.0:
            ratios.append(float('NaN')) #NaN = Not a Number
        else:
            ratios.append(vect1Elem/vect2Elem)
    return ratios

```

7 - 2 - 2 التحكم في التدفق بدون استخدام الالية try-except

الكود في الشكل 7 - 2 طويل وصعب قراءته أصعب من الكود الموجود في الشكل 7 - 1 وأيضا أقل كفاءة ، (لاحظ أن الكود في 7 - 2 قد تم اختصاره عن طريق تقليل المتغيرات المحلية مما جعله أقل كفاءة حيث يتم الوصول الى كل عنصر في كل تكرار). دعونا ننظر الى كود آخر:

```
def getGrades(fname):
    try:
        gradesFile = open(fname, 'r') #open file for reading
    except IOError:
        raise ValueError('getGrades could not open ' + fname)
    grades = []
    for line in gradesFile:
        try:
            grades.append(float(line))
        except:
            raise ValueError('Unable to convert line to float')
    return grades

try:
    grades = getGrades('quiz1grades.txt')
    grades.sort()
    median = grades[len(grades)//2]
    print 'Median grade is', median
except ValueError, errorMsg:
    print 'Whoops.', errorMsg
```

الشكل 7 - 2 - 3 جلب الدرجات

الوظيفة (`getGrades`) إما أن تعيد قيم أو تطلق استثناءات ، كل بقيمته الملحقة ، سوف تطلق الاستثناء `ValueError` اذا تم استدعاء الوظيفة `open` وأطلقت استثناء `IOError` ويتركه للجزء `getGrades` من البرنامج للتعامل معه ، لكنه سيقدم معلومات أقل عن الشيء الذى تسبب فى الخطأ

الجزء `getGrades` ينتظر القيمة العائدة لحساب قيمة اخرى أو القبض على استثناء وطباعة رسالة خطأ نافعة.

7 – 3 التأكيدات Assertions

الحالة assert فى بايثون تمكن المبرمجين من تأكيد أن الحالى الحسابية مقبولة ، ويمكن كتابتها فى أحد الشكلين :

assert Boolean expression

أو

assert Boolean expression, argument

عندما يواجه البرنامج حالة تأكيد يتم تقييم التعبير المنطقى (اذا كانت القيمة تساوى true) سيتم متابعة التنفيذ أما اذا تم تقييم التعبير المنطقى الى false سيتم اطلاق الاستثناء AssertionError .

التأكيدات هى وسيلة جيدة للبرمجة الدفاعية ، ويمكن استخدامها للتأكد من أن مدخلات وظيفة ما من النوع الصحيح ، هى أيضا تعتبر أداة تصحيح فاعلة ، فيتم استخدامها (على سبيل المثال) للتأكد من تمرير قيم صحيحة أو أن الوظائف تعيد قيم صحيحة