RAK-19006 Various Topics of Civil Engineering: **Python 3 for scientific computing**
Spring 2018, TUT, Tampere

Published 14.2.2018

Solutions for exercises, *lectures 1–2*

[These example solutions contain additional information – don't worry if you haven't thought of everything. I hardly expect anyone who has not used Python before to know about exotic corner cases, internal workings, and similar. Same with any points on SW engineering; needs exposure and practice.]

1. ★☆☆ *Python's approach to variables.*

   a) On the first line, a new name “x” is defined, and bound to an *int* instance that has value 1.

   On the third line, the same name is re-bound to a new value. The right-hand side is evaluated first, to determine this new value. We take the value currently pointed to by the name “x” (i.e. 1), and add 1. The resulting object (an *int* instance that has value 2) is then bound to the name “x”.

   Hence, *the printed ids are different*, because at the two moments in time when the id is printed, the name “x” points to different objects. For illustration, see Ned Batchelder's essay *Facts and myths about Python names and values*.

   This is the general case. It is of course possible to construct paradoxes such as this:

   ```
   x = 1
   i1 = id(x)
   x = x + 1
   x = x - 1
   i2 = id(x)
   i1 == i2  # True?!
   ```

   But as always with paradoxes, it is an illusion. The paradox is resolved when we recall the observation that Python treats number instances as singletons; due to this, here the final result 1 must be the same object instance as the original 1.

   In conclusion, numbers in computers are a tricky thing:

   ```
   0.1 + 0.1 + 0.1 == 0.3  # False (try it!)
   ```

   ...so one needs to be careful also with numerical equality, not only object identity, and this is not specific to Python. We will dig into this when we talk about the behavior of floating point numbers later.

b) The result cannot change whether or not the shorthand notation is used, because Python's *int* instances are immutable. The value of an *int* instance *cannot change*; the existence of an alternative notation does not affect this.

If we want to go slightly deeper:

For ohjects in general, “x + 1” calls *x.__add__(1)*, whereas “x += 1” calls *x.__iadd__(1)* if that is available (and redirects to *__add__* if not). Hence, what exactly happens, depends on the details of the implementation of each particular type of object.

However, with the focus on clarity in the Python community, it is perfectly reasonable to expect objects to behave sensibly. Mutable objects might (or might not) have a specialized *__iadd__* method that modifies the object instance in-place. Very much in the spirit of duck typing, immutable objects simply don't provide that. When someone “+=”s an object that doesn't, Python notices that, uses *__add__* to make a new instance for the result, and then re-binds the name, just like for *int* in item a) here.

c) Instead of names and values, the C language has variables. Here is the corresponding example program in C:

https://github.com/Technologicat/python-3-scicomp-intro/blob/master/lecture_slides/sol1_1c.c

Example output:

```
0x0x7fff90efbd7c
0x0x7fff90efbd7c
```

We see that the memory address remains the same. This is because a variable is a *name for a fixed storage location*. Here “x = x + 1” first computes the new value (just like in Python), and then (unlike in Python!) writes it into the location “x”, overwriting the original value.

d) As is easily tested, the program prints

```
['testing', 1, 2, 3]
```

The main point here is that in Python, **assignment never copies**; so “B = A” will simply create a new name “B”, pointing to the same object instance.

Of course, the *name* “B” is independent of the *name* “A”; we could now re-bind “A” to mean something else, and “B” would be unaffected.

Here Python's help can be handy for looking up what methods *list* has, as also for seeing what arguments they take and in which order (if not possible or sensible to pass by name).

In particular, the first argument of *list.insert* is the position *before which* the new item will be inserted.

2. ★☆☆ *Python's documentation.*

a) Use an empty string for the *end* keyword argument to **print**:

```
print('This string is printed without a newline', end='')
```

The *sep* kwarg is also useful; it can be used for things like this:

```
lst = [1, 2, 3]
print(*lst, sep='; ')
```

which may save some typing effort if we want to e.g. debug-print the contents of a list, using a custom separator.

Of course, in this particular case we could equivalently:

```
lst = [1, 2, 3]
s = '; '.join(str(x) for x in lst)
print(s)
```

Which is a bit longer, but has the advantage that the string we build like this can be e.g. passed into a logger (or into other things that do not come with a “sep” option).

The generator expression is needed (we cannot just say ...join(lst)), because join expects a list of *str* as its input. (Strong typing of values!)

b) The counterpart of **print** is called **input**. It can be found in the builtins section of Python's documentation (which was linked in the question).

For example:

```
username = input('Who are you? ')
msg = 'Hello, {:s}!'.format(username)
print(msg)
```

※ Be careful when reading old code. In Python 2, **input** was a different built-in function that executed its input as Python code (generally a catastrophically bad idea!). What Python 3 calls **input** used to be called **raw_input**.

If you really, really want to execute user input as code (and are aware of the security implications), you still can: just exec() or eval() the input as appropriate.

3. ★☆☆ *List and string operations.*

a) Use a negative step. (Default begin and end will do.)

```
s = 'sdrawkcab regnol on si gnirts siht'
t = s[::-1]
```

b) We have:

```
lst = [1, 2, 3]
lst.append([4, 5])
```

Each of the first three elements in *lst* has type *int*. The fourth element has type *list*, and it contains two elements of type *int*.

If we needed to do a lot of this kind of analysis, we could automate it e.g. as follows:

```
def analyze(L, indent=0):
    s = indent * " "
    for k,x in enumerate(L):  # built-in, see also help(enumerate)
        print('{:s}{:d} {} {}'.format(s, k, type(x), x))
        if isinstance(x, (list, tuple)):  # multiple types means “one of”
            analyze(x, indent+4)
```

Now, when we

```
analyze(lst)
```

we obtain the output

```
0 <class 'int'> 1
1 <class 'int'> 2
2 <class 'int'> 3
3 <class 'list'> [4, 5]
    0 <class 'int'> 4
    1 <class 'int'> 5
```

c) This will do nicely:

```
import math
s = '{:0.100f}'.format(math.pi).strip('0')
print(len(s) - 2)  # 2 for the initial “3.”
```

I got 48 nonzero decimals on CPython 3.4.3. How many of them are correct, and how many come from rounding to the nearest representable number, is another matter. Also, even if all of them were correct, we are here assuming that the final known decimal is not a zero.

This is of course silly; the main points here are the string formatting syntax (on which see the links on slide 19 in lecture 2), and that *strip* has an optional argument. While *strip* is mostly used for chopping off leading/trailing whitespace, it can also remove other leading/trailing characters. (It also has cousins *lstrip* and *rstrip* for one-sided stripping.)

The likely slower, **while** loop solution:

```
import math
s = '{:0.100f}'.format(math.pi)
while s.endswith('0'):    # can also use s[-1] == '0' instead
    s = s[:-1]    # take all but last
print(len(s) - 2)
```

※ Using the arbitrary-precision library mpmath to actually get 100 decimals of π:

```
import mpmath
mpmath.mp.dps = 100
pi = mpmath.pi()
print(str(pi))
```

For the record, the result is (actually 99 decimals; on mpmath version 0.19, bumping to 101 significant digits does not help; bumping to 102 gives 101 decimals):
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 8

d) The result of "a,b = b,a” is to swap where the names “a” and “b” point to; this is called a ***pythonic swap***.

There is no need to introduce a temporary to perform the swap, unlike in many other languages.

Python essentially packs the right-hand side into a tuple, and then unpacks it to the names on the left-hand side. It is guaranteed that the assignments happen left-to-right (which may be worth keeping in mind in case you want to overwrite into the same name for some reason).

Two- and three-operand cases are special-cased in the Python virtual machine for faster operation. With four or more operands, the tuple is explicitly constructed.

For the adventurous, a more detailed explanation and bytecode at StackOverflow.

4. ★★☆ *Flow control; comprehensions.*

a) Here is a version using **for** and **if**:

```
out = []
for k in range(100):
    m = k**2
    if m % 3 == 0:
        out.append(m)
print(out)
print(len(out))
```

The length of the output is 34 (the sequence starts at the value 0).

b) List comprehension, two lines (plus the printing):

```
sqrange = (x**2 for x in range(100))  # generator expression, we don't need all at once
out = [m for m in sqrange if m % 3 == 0]
print(out)
print(len(out))
```

We may shorten this to one line (plus the printing) by inlining the generator expression:

```
out = [m for m in (x**2 for x in range(100)) if m % 3 == 0]
print(out)
print(len(out))
```

c) For programmers coming from most languages, the natural first solution is perhaps something like:

```
def sum_neighbors(lst):
    out = []
    for k in range(len(lst) - 1):
        out.append(lst[k] + lst[k+1])
    return out

lst = range(10)
print(list(lst))
print(sum_neighbors(lst))
```

However, indexing an iterable in this manner, when what we want to do is to operate on its elements, is not very pythonic.

Instead, we may operate directly on the elements by slicing and zipping, which makes the intent of the code clearer:

```
def sum_neighbors(lst):
    out = []
    for a,b in zip(lst[:-1], lst[1:]):
        out.append(a + b)
    return out
```

Since the documentation tells us that zip stops when its shortest input is exhausted, we don't strictly need to cut elements at the end. Hence, we could as well write:

```
def sum_neighbors(lst):
    out = []
    for a,b in zip(lst, lst[1:]):
        out.append(a + b)
    return out
```

Taking advantage of list comprehension, this is shortened to:

```
def sum_neighbors(lst):
    return [a + b for a,b in zip(lst, lst[1:])]
```

※ But how to generalize this, if we want to sum more than two adjacent elements?

On the internet, Scott Triglia has posted a solution to a very similar problem:

http://locallyoptimal.com/blog/2013/01/20/elegant-n-gram-generation-in-python/

So the final ingredient – to programmatically build the input for zip – is:

```
[lst[i:] for i in range(n)]
```

This returns a list, which we may unpack to **zip** (like we did manually for two entries, above). During each iteration, the zip then returns a tuple of elements that are to be summed.

Because *n* is only available at runtime, we cannot name the elements (like “a” and “b”, above); instead, we will capture the whole tuple as “item”. To sum its elements no matter how many, we can use the **sum** built-in, which takes an iterable. (To do some other operation instead, we would create a function taking in **args*, and then operate on those.)

Final result, generalized for summing over *n* adjacent elements:

```
def sum_neighbors(lst, n):
    return [sum(item) for item in zip(*[lst[i:] for i in range(n)])]
```

Testing it:

```
for k in range(len(lst)+1):
    print(sum_neighbors(lst, k))
```

d) This inverts the input dictionary, provided that the inverse exists:

```
def invert_dict(dic):
    return {v: k for k,v in dic.items()}

dic = {1: 'a', 2: 'b', 3: 'c'}
inv = invert_dict(dic)
print(dic, inv)
```

e) *Order-preserving uniqification.*

First, here is the straightforward solution that probably first comes to mind.

We keep a set of items that have been seen, and then filter based on whether each item is already in that set. The helper function *is_unique* updates the set, and returns whether the item was already there. (When calling *set.add*, strictly speaking it doesn't matter whether the item is already in the set; recall that a set automatically discards any duplicates. Note, however, that the ordering of operations does matter: we must first check, then update.)

```
def uniqify(lst):
    seen = set()
    def is_unique(x):
        if x not in seen:
            seen.add(x)
            return True
        else:
            return False
    return (x for x in lst if is_unique(x))  # equivalent with filter(is_unique, lst)
```

Even though we return a generator expression (instead of immediately returning a list), the returned generator is a closure, so it will have its own copy of *seen*. (This observation is important if these generators for several different inputs are extracted in an interleaved manner.)

Testing:

```
lst = [1, 2, 4, 4, 4, 3, 1, 2, 2]
print(uniqify(lst))
```

The solution can be shortened by abusing the short-circuiting **or** to inline the uniqueness check:

```
def uniqify(lst):
    seen = set()
    return (seen.add(x) or x for x in lst if x not in seen)
```

The cryptic-looking ***or*** *x* is necessary, because *set.add* is very imperatively minded: it modifies the set in-place (which we do want), and *returns None* (which is unfortunate). Because **None** is falsey, **or** will have to evaluate its second subexpression – in this case, *x* itself. Hence, "*seen.add(x)* ***or*** *x*" really means "add *x* to seen (if not already there), and also return *x*".

A one-liner is possible, at the cost of any semblance of readability:

```
def uniqify(lst):
    return (lambda seen: (seen.add(x) or x for x in lst if x not in seen))(set())
```

This looks unnecessarily cryptic, because Python does not have a Lisp-style **let** form. If it did, we could write (**this is not valid Python**):

```
def uniqify(lst):
    let [seen = set()]:    # ← Python is missing this syntax
        (seen.add(x) or x for x in lst if x not in seen)
```

A Lisp-style **let** is really just a rearranged **lambda**, that is immediately called with some values:

let [a = foo, b = bar] (…) ⇒ (**lambda** a, b: …)(foo, bar)

The **let** form tends to be much more readable (because the names and values appear together), especially if the *body* (the "…" above) is long.

So, the one-liner is defining a function of the parameter "*seen*", which is then immediately filled with a new, empty set. Because the empty set is an argument, not a default value, a new empty set is created every time *uniqify* is called.

Of the solutions Python does support, which one is the most readable?

This is a matter of opinion. Personally, I find the straightforward one far too verbose, although it certainly looks like it would probably appeal to a large subset of the Python community.

I prefer the two-line version – although we must abuse **or** to do it (because *set.add* is too imperative – why doesn't it just return the value that was added?), it is still small and simple enough to parse with one reading (assuming some previous experience doing this kind of thing).

5. ★★☆ *Functions.*

a) The code, repeated here for convenience:

```
def f(a, b, *args):
    print('=' * 20)
    print('a = {:s}'.format(str(a)))
    print('b = {:s}'.format(str(b)))
    for x in args:
        print('extra: {:s}'.format(str(x)))
f(b=1, a=2)
f(1, 2, 3)
```

The code prints

```
====================
a = 2
b = 1
====================
a = 1
b = 2
extra: 3
```

The first call passes arguments by name; hence their ordering does not matter.

The second call passes by position. Because the function “f” only takes two named arguments, the extra one goes into the list *args*. Hence, *args* will be a one-element list, which is then processed by the **for** loop to print out the extra argument.

If we try this:

```
f(b=1, a=2, 3)
```

Python raises SyntaxError, where the error message tells us that if we want to pass arguments by name, those must come **after** any positional arguments. This is just a syntactic convention.

b) Slides 30–38 in lecture 2 provide examples of the different argument passing styles.

[Note: there is a mistake on slide 32: formatting should be {}, or alternatively the argument to format() must be explicitly converted to str: *format(str(a))*, and similarly for *b*. The same applies to the examples on slides 35–38.

This bug slipped through, because I'm (far too) used to the old string formatting with the % operator, where %s converts anything into a string. Most of the code examples have been actually tested, but it seems I forgot to test these particular ones! Sorry about that.

The newer str.format and f-strings are both more pythonic: they enforce the strong typing of values. Using empty curly braces {} removes the type restriction, behaving like the old %s.]

c) Here is **filter** in pure Python (to complement **map**, on slide 41 in lecture 2), with an example:

```
def filter(pred, lst):  # LEGB, doesn't matter that the name is already taken
    out = []
    for x in lst:
        if pred(x):
            out.append(x)
    return out

def evenp(x):
    return x % 2 == 0

lst = range(10)
print(filter(evenp, lst))
```

d) It is easily observed, by testing, that the output is:

```
9 125 (0, 4, 16, 36, 64) (1, 2, 3)
hello
```

Why? Consider the code again:

```
hello = lambda: print('hello')    # a lambda without parameters is also fine

f = lambda x: x**2
a = f(3)
b = (lambda x: x**3)(5)
pack = lambda *args: tuple(args)    # adaptor; “multi-arg tuple constructor”

lst = range(10)
out = tuple(filter(lambda x: x % 2 == 0, map(lambda x: x**2, lst)))

print(a, b, out, pack(1, 2, 3))
hello()    # function call with no arguments
```

Important points:

- A **lambda** without parameters ***delays evaluation***. It is also called a *thunk [FP]*. The expression is evaluated when the thunk is called. Hence, “hello” prints last.
- In principle, **def f**(x): (…) ⇒ f = **lambda** x: (…). Python's lambda has limited functionality, but Lisp indeed expands function definitions like this.
- On the line “b = …”, we create, apply, and discard an anonymous function.
- Python's *tuple* constructor supports only a single argument, which must be an iterable. To take individual function arguments and pack them into a tuple, an adaptor such as “*pack*” here can be used. Often this is not needed, since one can just *(a, b, c)* instead, but there are some exotic cases where this is useful (example in week 10).
- A list comprehension combines *filter* and *map*, but unlike here, *filter* first, then *map*. This code squares each input, and then accepts only the even results. Generally, if possible, it is better to filter first, because often the map step is more expensive.

6. ★★★ *Generators; infinite sequences.*

a) Here is a generator for *n* first nonnegative integer multiples of 3:

```
def multiples_of_3(n):
    c = 3
    end = c*n    # in Python, no keyword end, so available as a variable name!
    k = 0
    while k < end:
        yield c*k
        k += c
```

The closure property is useful here, because it makes the generator remember value of *n* that was originally passed in when the generator was first invoked.

Testing:

```
for x in multiples_of_3(10):
    print(x)
```

b) Natural numbers, starting from a given *n*, are generated easily enough:

```
def naturals(n=0):
    k = n
    while True:
        yield k
        k += 1
```

Note that this generator never finishes; it represents a countably infinite sequence.

Getting the first 100, and all under 200 that are divisible by 7:

```
from itertools import islice, takewhile
lst = islice(naturals(), 100)  # this is still just a promise!
print(tuple(lst))              # use tuple() to force the promise
divisibles = [x for x in takewhile(lambda k: k < 200, naturals()) if x % 7 == 0]
```

The important points are that *islice* slices an iterable (possibly infinite), and *takewhile* allows to test each extracted element, and stop extracting when the given predicate becomes falsey.

For a more advanced application (still a teaching example!), see these prime number sieves:

https://github.com/Technologicat/python-3-scicomp-intro/blob/master/examples/sieve.py

The Sieve of Eratosthenes one is based on SICP, 2nd ed., section 3.5.1. The main idea is: taking the natural numbers starting from 2, the first element is a prime. Add a filter to the remainder of the stream to remove its multiples; then the first element is a prime. Add another filter…

c) A first attempt at improving the robustness is:

```
def naturals(n=0):
    if not isinstance(n, int):
        raise TypeError('expected nonnegative integer n, got {}'.format(type(n)))
    if n < 0:
        raise ValueError('expected nonnegative integer n, got {}'.format(n))
    k = n
    while True:
        yield k
        k += 1
```

Considering the fail-fast principle, the remaining problem is this:

```
Z = naturals(-10)    # this should fail, right?
for x in Z:
    print(x)
```

Run this. Observe *at which line* the error is triggered.

In a real-world application, it is not unimaginable that the stream is first generated somewhere, and then extracted somewhere else (much) later. To have any chance at debugging, we should fail fast, before the context where the error originated is lost; after all, we have enough information to detect and report the error already when the stream is created.

The problem is that the generator *does not yet run when invoked*: the function body is only entered when the first element is requested. How to work around this?

One solution is to wrap the generator definition, like this:

```
def naturals(n=0):
    if not isinstance(n, int):
        raise TypeError('expected nonnegative integer n, got {}'.format(type(n)))
    if n < 0:
        raise ValueError('expected nonnegative integer n, got {}'.format(n))
    def gen(n):
        k = n
        while True:
            yield k
            k += 1
    return gen(n)  # invoke the generator (to create the iterator, as usual)
```

That is, do the checking first in a regular function, and only then proceed to invoke the generator. Invoking the wrapped generator makes the wrapper function itself appear to the outside world as if it was a generator. Testing again:

```
Z = naturals(-10)   # kaboom, ValueError, good!
Z = naturals(0.5)   # kaboom, TypeError, great!
print(tuple(islice(naturals(n=5), 10)))  # OK [remember to test also for regressions!]
```