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

Published 7.2.2018

Exercises, *lectures 1–2*

Meta:

- Since all of you come from a wide variety of backgrounds, the idea is *something for everyone*.

  The exercises are intended to support learning; not to pile on superfluous trivialities, nor on the other hand to raise blood pressure by providing completely insurmountable obstacles.

  If a problem feels too simple or too advanced, skip it – pick what you think is appropriate. Difficulty levels are marked with a star rating from 0 to 3.

- One week to solve.
  - We will discuss solutions the following week, provided that time allows.
  - In any case, solutions to many (but not necessarily all) questions will be sent via email to course participants.

- Submitting solutions is not compulsory, but if you want feedback, you can (by email).

- In de facto standard university style, there will be 5–6 questions per week. (But each may have several subitems.)

- Any provided code is copy'n'pasteable. (If it is not, that's a bug; email me and we'll fix it.)

Questions, *lectures 1–2*:

0. ☆☆☆ *Minimal background questionnaire.*

   [Please submit an answer to this – the information will be used to tune upcoming exercises!]

   ◊ Which programming languages are you familiar with, at least to the extent that you recall the related concepts? E.g. “C, C++, Fortran, MATLAB”.

   ◊ At what level – approximately – do you know the programming language(s) you mainly use? A one-word answer such as basic / intermediate / advanced is fine, or if you feel you're between two levels, feel free to say so.

   ◊ If you have developed software, was it numerical solvers (i.e. the data-processing core functionality only), command-line applications, GUI applications, web applications, other/please specify?

   ◊ Previous knowledge of CS (computer science)? Are you familiar with data structures (e.g. stack, heap, tree, graph), algorithms, programming paradigms, etc.?

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

   a) Consider the following code:

```
x = 1
print(id(x))
x = x + 1
print(id(x))
```

   Assume `x` is not defined before we run this. What happens on the first line? On the third line?

   Are the printed ids the same or different? Why?

   b) Does the result change if you use shorthand `x += 1` instead of `x = x + 1`? Why or why not?

   c) If you are familiar with the C language, how does the result differ if you were to perform the same sequence of operations in C? Why? (Assume `x` is declared as an int.)

   d) What does the following program print, and why?

```
A = [1, 2, 3]
B = A
B.insert(0, 'testing')
print(A)
```

   (If needed, recall that you can call `help(list.something)` as well as `help(A.something)`, where `A` is a list object instance; both will work.)

2. ★☆☆ *Python's documentation.*

   The official documentation has a section that lists and describes all built-in functions in Python:

   https://docs.python.org/3/library/functions.html#built-in-functions

   Take a look at the documentation.

   a) How do you print a string without causing Python to add a newline?

   b) Write a short program (~3 lines) that lets the user input a string, inserts the result into another string (via string formatting), and prints the final result. (E.g. the classic “Hello, <name>!”)

   ※ The main purpose here is to get a rough idea of what Python provides as built-in; also, it would feel silly to introduce print(), but not its natural counterpart.

   In practical numerical code, solver parameters will usually be hardcoded just like in MATLAB scripts, read in from a file (see **open**), or passed in from the command line (see argparse in the standard library).

3. ★☆☆ *List and string operations.*

   a) How do you reverse a string, using only slicing?

   b) Consider this code:

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

   Whoever wrote that clearly intended to .extend(), not .append(). With the current code, what can you say about the types of the elements in lst? (Use indexing and the built-in **type** to examine.)

   c) See the documentation on Python's string formatting (links provided in lecture 2).

   import math, and print math.pi to 100 decimals using string formatting (pick any of the three variants of string formatting syntax).

   Chop off any trailing zeros from the string (in Python, not manually!). How many decimals did you actually get? To get the number of elements in a container, use the built-in **len**.

   Hint: str.strip; alternatively, str.endswith, slicing, and a **while** loop.

   ※ mpmath provides a way to actually get π to arbitrary precision.

   d) Below, what is the result? Why? (*Hint: on the third line, the RHS behaves like a tuple.*)

```
a = 2
b = 3
a, b = b, a
```

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

a) Using **for** and **if**, write a short program that takes the range of integers from 0 to 99, squares each number, and accepts only those results that are divisible by 3. Place the results into a list. How many are there?

The modulo operator in Python is %; equality comparison is ==.

b) Let's make the program even shorter. Express the same operation using list comprehensions (one or several; either is fine).

c) Write a short function that takes a list of numbers, sums each pair of two consecutive elements (elements 0 and 1; 1 and 2; 2 and 3; ...), and returns the results as a new list.

How to loop over the pairs of items directly, without using an index as the loop counter? (*Hint: slicing*, *zipping*.)

Test your solution on some input; the input can be hardcoded, e.g. `lst = range(10)`.

d) You have a dictionary:

```
dic = {1: 'a', 2: 'b', 3: 'c'}
```

In this particular case, the values happen to fulfill the conditions required of keys: they are immutable and hashable. They are also unique, so an inverse dictionary exists.

Write a dict comprehension that inverts its input, i.e. produces a new dictionary that looks up items in the opposite direction.

e) *Order-preserving uniqification.*

In lecture 2, it was noted that if you have a list (of hashable items) with duplicates in it, you can easily use a `set` to extract the unique items, if the ordering of the result does not matter:

```
lst = [1, 2, 4, 4, 4, 3, 1, 2, 2]
uniques = list(set(lst))
```

We first create a set, which we then convert back into a list (if we want a list as output).

But how to extract the unique items, while preserving *the ordering in which each unique item first appears* in lst?

(Use either a loop or a list comprehension; either approach is fine. How short can you make the code? If you try several solutions, which approach is the most readable?)

5. ★★☆ *Functions.*

a) What is printed by the code below? Why?

```
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)
```

What happens if you try to run this?

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

b) Implement a few small functions, trying out the different variants of argument passing that were introduced in lecture 2.

c) Implement the higher-order function *filter*. Its parameters are a predicate, and a list. It should inspect each element of the list, in order, and return a new list of those elements that satisfy the predicate.

The predicate, to be provided by the user when calling filter(), is a function of one argument – the element being tested – that returns truthy if filter() should let that element through to its output.

Create a list, a predicate, and test your implementation with them.

※ Python provides **filter** as a built-in; and list comprehension is preferred for that task. This is just meant as a look inside a higher-order function.

d) What does this code print?

```
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
```

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

On lecture 2, when talking about list comprehensions, we briefly mentioned *generator expressions*, which are essentially lazy comprehensions. Each element is evaluated as it is requested, rather than all elements at once.

Generators in Python are more general than this. Using **yield** instead of **return** turns a function definition into a generator (instead of a regular function).

The main points of a generator are:

◊ When a generator is called, it does not yet run; instead, an iterable is returned to the caller.
◊ Calling next() on that iterable (i.e. iterating on it) will then actually start the generator.
◊ After reaching **yield**, the generator is suspended, returning the yielded value.
◊ When next() is called on it again, execution continues from where it left off – remembering the state.
◊ Once all elements have been yielded, the generator raises StopIteration (which is the mechanism Python uses to signal a **for** loop to exit).

See:
https://stackoverflow.com/questions/1756096/understanding-generators-in-python

a) Armed with this, implement a generator that generates nonnegative integer multiples of 3, either until the result reaches a given number, or up to a given number of terms (your choice).

(Use a **while** loop in the generator. To parameterize, note that when the generator is called, it will return a closure.)

b) Generators can be used to generate infinite sequences, finite parts of which can then be actually evaluated. Write a generator for natural numbers, starting at a user-given nonnegative integer *n*.

Look up islice and takewhile in the itertools module. With your generator:

◊ Get the first 100 natural numbers.
◊ Take all natural numbers under 200 that are divisible by 7. (Filter using a list comprehension.)

※ The itertools module, in general, may be worth a look when working with iterables.

c) Let's make the generator more robust. Validate that the input *n* is a positive integer, raising TypeError or ValueError as appropriate, if at least one these conditions is violated.

However, considering the fail-fast principle, this leaves a problem. Spot the issue and fix it.