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

Published 14.2.2018

Exercises, *lecture 3*

1. ★☆☆ *Imports.*

   a) Consider the following example, consisting of three files (placed in the same folder):

```
# a.py
def hello():
    print("hello from a.py")

# b.py
def hello():
    print("hello from b.py")

# main.py
from a import *
hello()
from b import *
hello()
```

   What is printed when we run *main.py*? Why?

   Reorganize *main.py*, placing both imports at the top. Call both versions of hello().

   b) Extract the Unicode names of the following characters, using Python.

```
lst = ['ℝ', '数', 'ℓ', '⁂', 'あ', 'א', '⇨', '◊', '※', '♫']
```

   Also available as a .py file here, may work better than copy'n'pasting from the .pdf:
   https://github.com/Technologicat/python-3-scicomp-intro/blob/master/lecture_slides/ex3_chars.py

   c) We have already met the *math* module. Convert this snippet into working Python code:

```
out = []
n = 10
for k in range(n+1):
    x = k/n * 2*pi
    y = sin(x)
    out.append((x,y))
print(out)
```

   ※ Actual numerics code uses np.linspace and vectorized math, to be covered next week. Whatever toaster your code runs on, it will always have at least NumPy.

d) Colors have many different representations. Conversions between them can be useful.

For example, the HSL representation (which Python calls "HLS") provides a way to parametrically generate different brightnesses that have the same hue; or walk through hues like in a rainbow, while keeping the same brightness and saturation. This can be useful in visualizations. However, the commonly used input format is RGB.

See the *colorsys* module (for Python **3**), and convert the RGBA color '#6fa3f0ff' into HLS. Here the input format is #rrggbbaa (the hash is literal; then two hex digits per component); discard the alpha (opacity) component.

You will first need to extract the R, G and B components from the input string, and convert them into the format the appropriate conversion function in *colorsys* expects.

(*Hint: the int constructor.*)

> ※ HSL is not perceptually accurate (relevant XKCD), but often good enough. There are color systems specifically designed to take into account the behavior of the human visual system, such as CIELAB. On the topic of data visualization, How bad is your colormap? (If it is, one solution for that is colorcet.)

2. ★☆☆ *Of Unicode sandwiches and I/O.*

Download this data file (utf-8):

https://github.com/Technologicat/python-3-scicomp-intro/blob/master/examples/data/utf8_text_file.txt

a) Using **with**, **open**, and a **for** loop, read the file line-by-line into a new list. Print the list to verify it loaded correctly.

(If you have a different font and some characters are not available, it doesn't matter, as long as most of the file looks fine. But pay attention to äöå; if the encoding is wrong, scandinavic characters will reveal that, since they will then show as gibberish.)

b) Modify your program:

Append, insert, replace or delete some lines after the list is loaded. Save the result under a different filename (similarly using **with**, **open**, and a **for** loop).

For saving, the **open** mode is 'wt' for "write text". As for how to write into the file, invoke help on the file object you get from **open**; you can do that already when you have the file open for reading (e.g. in the REPL!). Alternatively, look at help(open); the types it can return are documented near the end. (Yet alternatively, search the internet.)

c) Replace the **for** loop reading the file with a list comprehension. In your opinion, does the resulting code look more readable, or less?

> ※ In numerical code, direct file access with **open** is rather rare, but it is useful for other use cases. For numerical data, we will cover typical save/load approaches later.

3. ★★☆ *Language features.*

[Here the explanations may be long, but each item is short to do.]

a) *Acronym fight! LBYL vs. EAFP*

To look at error handling approaches, we will use a technique known in unit testing as ***mocking***, where we replace irrelevant parts of the system by *mock objects*. The terminology comes from object-oriented programming, but the idea applies generally.

A mock object emulates the interface of "the real thing", but provides only the minimal functionality that is needed to perform the test. (In unit testing, the mock object represents a part of the system that is external to the unit being tested., but that's immaterial here.)

Since we concentrate on error handling, we need a function that can succeed or fail. For the *Look Before You Leap (LBYL)* paradigm, the function should return truthy on success, falsey on failure. In the real world, it would have some side effects (including output arguments) to perform useful work, but we can ignore that in the mock. Here is what we will need:

```
import random

def random_failure(p):           # p ∈ (0,1); probability of failure
  if random.uniform(0, 1) < p:  # failed?
    return False
  else:
    return True

class DeterministicFailure:    # doing this needs state, so let's use a class
  def __init__(self, n, k=0):
    self.n = n          # every nth call will fail
    self.k = k % n    # 0, 1, …, n-1; the index of the current call
  def __call__(self):    # magic method that makes instances of this class callable
    oldk = self.k
    self.k = (self.k + 1) % self.n
    if oldk % self.n == 0:
      return False
    else:
      return True
```

Armed with this:

- Create a function, and in it, perform three randomly failing operations. Each time, check the return value , and use **if** statements to decide whether your function should proceed, or finish by **return**. Avoid nesting. (*Hint: organize appropriately;* ***else*** *blocks are not strictly needed for this.*)
- Try the same with the deterministically failing computation. The class behaves like a function factory; *f = DeterministicFailure(3)* will return a function instance that fails every third time it is called. The call syntax is just *f()*, since the only parameter is *self*.

So, that's how LBYL code typically looks – assuming that the programmer has at all bothered to implement error handling. (If not, the end result is anything but robust!)

Next, let's try *Easier to Ask for Forgiveness than Permission (EAFP)*, using exceptions. We will need to change our mock objects to conform to this paradigm, since the semantics are different: the return value no longer signals errors, and (in the real world) can be used to pass data. (Note especially that since the return value can be any Python object, including containers – and **return** does not make a copy – most often there is no need for output arguments!)

Here are the updated mocks:

```
import random

def random_failure(p):            # p ∈ (0,1); probability of failure
   if random.uniform(0, 1) < p:  # failed?
      raise RuntimeError('Random failure triggered, p = {:g}'.format(p))

class DeterministicFailure:    # doing this needs state, so let's use a class
   def __init__(self, n, k=0):
      self.n = n          # every nth call will fail
      self.k = k % n    # 0, 1, …, n-1; the index of the current call
   def __call__(self):    # magic method that makes instances of this class callable
      oldk = self.k
      self.k = (self.k + 1) % self.n
      if oldk % self.n == 0:
         raise RuntimeError('Deterministic failure triggered, n = {:d}'.format(self.n))
```

Note especially that the paradigm change simplified the structure of the mocks; the **else** blocks have vanished. Often, if a paradigm change makes code shorter, it indicates that the new paradigm is a better fit to the structure of the problem.

The above however pales in comparison to the simplification at the calling end:

- Create a function, and in it, perform three randomly failing operations. Use the **try**/**except** construct to catch *RuntimeError* (catch no other exceptions!).
- Do the same for the deterministically failing computation, but this time, make the *second* call fail by setting the initial value of the counter *k* to *n* - 1 = 2 when you create your instance of the *DeterministicFailure* object.

Having done that, here are the important questions:

- What is the fraction of program length (lines of code) that performs error handling in each of the paradigms? (Some rough approximation is fine.)
- Concerning the handling of a specific kind (type) of error: if there are several places in the computation that may trigger it, what do you notice about the error handling code for it in each of the two paradigms?
- If the programmer does not bother to catch errors, do the resulting programs behave similarly, or differently, in the two paradigms? If differently, how? (Test, or deduce; either way is fine.)

b) *Lexical vs. dynamic scoping.*

Python uses lexical scoping, but it is possible to emulate dynamic scoping. For that, we have this StackOverflow answer packaged into a module *dynscope.py*, available for download here:

https://github.com/Technologicat/python-3-scicomp-intro/tree/master/examples/dynamic_scoping_emulation

For usage instructions, see *main.py* in the same folder, and help(dynscope) once downloaded.

Modify both example programs from lecture 3, slide 26, to use dynamically scoped variables as provided by *dynscope.py*. Verify that the second program now works.

Does the first program still work? Why or why not?

(*Practical hint: save all three programs into the same folder; easiest for* ***import****ing dynscope.)*

※ This can be useful if you have several layers of functions, and you need to pass in some data to the innermost layer, and don't really want to modify the parameter list at each level. (Plotter options in a heavily structured solver come to mind – set at the outermost level, needed at an inner level.)

Dynamically scoped variables are here a cleaner solution than lexical global variables, because they will go out of scope (and hence, automatically vanish from polluting the global namespace) as soon as the relevant code block exits.

Also, they are easily shared between functions in different modules, if both modules **import** dynscope. (But if you do this, be sure to document it in a comment; otherwise it will be almost impossible to later know where the values for those variables come from. This is one reason why lexically scoped variables are almost always better.)

c) *Duck typing. [To analyze this code, keep in mind lectures 2–3, and* ***callable*** *from ex. 3.]*

Consider this program:

```
def f(a):
    print(a)

def g(a, b):
    print(a, b)

class Frizzle:
    def __init__(self, a):
        self.a = a
    def __call__(self):
        print(self.a)

class Frobozz:
    def __init__(self):
        self.count = 0
    def __call__(self, a):
        self.count += 1
        print(a)

# usage: f = Counted(f)
# f( ...args of original f here… )    # call as many times as you want
# print(f.count)    # how many times has f been called?
class Counted:
    def __init__(self, f):
        self.count = 0
        self.f = f
    def __call__(self, *args, **kwargs):    # [extra] explain each *, ** in this method
        self.count += 1
        return self.f(*args, **kwargs)

def main():
    for func in (f, g, Frizzle(23), Frobozz(), Counted(f)):    # what exactly is each item?
        try:
            func(42)    # ☆
        except TypeError as e:
            print('{:s} fail: {:s}'.format(*(str(x) for x in (func, e))))    # (maybe bad style?)
        else:
            print('{:s} pass'.format(str(func)))

main()
```

In terms of duck typing, we can think of the function call marked with ☆ (in main())
as expecting *func* to be a duck that is callable, and takes one positional argument.

Which of the invocations succeed and which fail? Why?

d) *Call-by-sharing.*

Consider the following program:

```
class A:
    def __init__(self, a):
        self.a = a

def replace(obj):
    obj = "Hi, I'll be replacing obj"

def imperative_update(obj):
    if not isinstance(obj, A):
        raise TypeError("Expected an 'A' instance, got '{:s}'".format(str(type(obj))))
    obj.a = 2 * obj.a

def functional_update(obj):
    if not isinstance(obj, A):
        raise TypeError("Expected an 'A' instance, got '{:s}'".format(str(type(obj))))
    out = type(obj)(None)
    out.a = 2 * obj.a
    return out

def main():
    o = A(42)
    imperative_update(o)
    o = functional_update(o)
    replace(o)

main()
```

Questions:

- Is the *obj*, that each function receives, the same instance that is created in main()? (Deduce, or add appropriate prints and test.)
- Does replace() do what its name suggests? Why or why not?
- Does imperative_update() work as expected? Why or why not?
- What are the main differences between imperative_update() and functional_update()? In the definition? In the usage (i.e. at the call site)? Conceptually?
- In functional_update(), why is *type(obj)(**None**)* better than *A(**None**)*? *(Hint: OOP.)*
- What happens if you accidentally *type(A)(**None**)* instead of *type(obj)(**None**)*? Explain the error message. *(Hint: what is the value of type(A), and why?)*

4. ★★☆ *What you see is what you __get__():* descriptors and properties.

   Instance data attributes in Python **class**es have no need for accessor (getter) and mutator (setter) methods (unlike in Java), because Python supports properties, which allow a "smart" data attribute (which executes some code when you set or get its value) appear to the outside world exactly as if it was just a plain data attribute (which has no code attached).

   Note that in Python, a plain data attribute can be effortlessly changed into a property after-the-fact, when the **class** is already in use (perhaps by programmers all the world over!) – and its use sites need no modification.

   Let's implement a simple example. Head over to the documentation:

   https://docs.python.org/3/library/functions.html#property
   https://docs.python.org/3/howto/descriptor.html#properties

   Then, take this example class:

```
class Triangle:
    def __init__(self, a, b, c):    # a, b, c: side lengths
        self.a = a
        self.b = b
        self.c = c
```

   a) Implement *perimeter* as a read-only property (i.e. having only a getter), which computes the perimeter at access time, i.e. when the value is read.

   Similarly, implement also *area* as a read-only property (use Heron's formula).

   Test your implementation, for example using this:

```
def main():
    T = Triangle(3, 4, 5)
    print(T.perimeter)    # at the use site, properties look like data, not functions!
    print(T.area)
    T.a,T.b,T.c = 1, 1, 2**(1/2)    # mutate the state
    print(T.perimeter)
    print(T.area)
main()
```

b) Let's reimplement the example using a different strategy. To avoid re-computing the perimeter and area each time they are requested, let's update them only when the length of a side changes.

> ※ Modifying the sides one at a time will make the triangle temporarily invalid, but let's not worry about that now. If you use **(1/2) to take the square root, it will work also for negative inputs. (*math.sqrt* accepts only positive inputs.)

Make helper methods that compute the perimeter and area from the current values of *self*.a, *self*.b and *self*.c, and store the results in instance attributes *self*.perimeter and *self*.area, respectively.

Make the constructor trigger the perimeter and area calculations.

Finally, reimplement the side lengths a, b and c as read-write properties. The getter should just return the underlying instance attribute, which can be private. The setter should change the private value, and then trigger the area and perimeter computations.

Test the new version, using the same test program as above.

c) In Python terminology, what is a *descriptor*? How are properties related to descriptors?
(This is not an essay question. Keep it short, main points only; ≲ 5 sentences is enough.)

[Descriptors will be needed in an advanced code-analysis question in a later set of exercises.]

5. ★★★ *Dynamic scoping emulation* – what's under the hood?

   Consider the dynamic scoping emulator:

   https://github.com/Technologicat/python-3-scicomp-intro/tree/master/examples/dynamic_scoping_emulation

   Take a look inside *dynscope.py* (the code itself is just ~25 lines).

   As the original author Jason Orendorff says, this approach *is intended as a compromise between "don't do that" and stack inspection*.

   a) What is the central insight, i.e. how is it even possible to build dynamic scoping on top of a lexically scoped language?

   b) What is the mechanism used to handle nested dynamic scopes? Do the read and write of dynamically scoped variables work the same way, or is there an asymmetry?

   c) What is the role of each of the two classes in the implementation?

   d) What role does each of the special (magic) methods play in the implementation?

   e) Does the emulator work correctly if called from several modules? From several threads? (Write a short test, or just look at the code and deduce what will happen; either way is fine.)

   f) Does the implementation currently provide a way to rebind a dynamically scoped name to point to a new value? If so, test it; if not, suggest how this feature could be added.

   This raises the question of whether the we should allow **nonlocal**-like rebinding (of names defined in an outer dynamic scope), or whether it is better to allow rebinding only for names defined in the current (innermost) dynamic scope. In your opinion, which option feels more pythonic? Why?

6. ★★★ *Method resolution order. Automated code analysis.*

   [This is not difficult to do, but may require a lot of reading depending on your background.]

   On the lecture, we briefly mentioned how Python handles the order in which the classes will be tried when looking up methods in multiple inheritance: the C3 linearization algorithm.

   Consider this teaching example that implements C3 in pure Python:

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

   The program takes in a Python source file, and computes the MRO of any classes defined in it.

   The analyzer only looks at the header of each class definition, and ignores everything else. The input is parsed using the *ast* module, which is how Python reads Python – the module provides runtime access to certain parts of the Python compiler.

   a) Roughly, how does the program work? Familiarize yourself with the basics of *argparse*, *logging*, abstract syntax trees, *ast*, and the visitor pattern. (Links provided in the source code comments.)

   b) Save the following example code into *testing.py* and feed it to the analyzer.

   What is the resulting MRO?

   How does it follow from the specification of the algorithm? Look at the output and follow what the analyzer is doing. (By looking at the code, or by using a debugger; either way is fine.)

   (*Hint: run in a terminal; IPython's %**run** magic may misbehave, at least in Spyder's REPL. Alternatively, if running in Spyder, use Run ▷ Configuration per file… to set the command-line options to the program so you can use Run ▷ Run and Debug ▷ Debug as usual.*)

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass
```

   (If you put the **pass** on a separate line, then Python expects it to be indented, but this one-line format is also legal, since the colon already separates the statements. This is very rarely used, because almost always it damages readability, but here this vertically compact format is more readable.)

   c) Take the more complex example from the Wikipedia page on C3 linearization (link above), and reformat the example as Python. Interpret “extends” as “inherits from”. Feed the example to the analyzer. Observe the output.

   d) Are there inputs for which linearization of the inheritance graph is impossible? If so, do these inputs have a common feature? Provide an example.

7. [extra] ★★★(★) *Episode II: Revenge of the generators*.

Download this anagram generator:

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

Read through the source code and analyze how the program works.

Note on collections.Counter: it essentially is a multiset; but there is no ready-made method to check whether a given multiset contains another given multiset. Hence the function has().

a) What is the strategy the program uses to split the input word length into admissible sets of subword lengths? How is the sequence originally generated? How is it transformed?

Would this strategy be acceptable for very large *n*? Can you think of a strategy with better asymptotic performance? (Either on the level of general principles, or a specific algorithm; either is fine.)

b) What is the exact sequence of computational steps meant by this almost-English line in the function **anagrams**?

```
if not all((length in words_by_length) for length in the_split):
```

(This mainly highlights that high-level languages such as Python can be succinct.)

c) Explain the magic: what makes the algorithm backtrack automatically, if no word can be made out of the remaining letters?

Note that the algorithm may need to backtrack several levels at once (in a split with many components), if an early decision already implies a dead-end for the remaining (multi-)set of letters. How does the code implement this?

d) *[optional]* Do you spot opportunities for refactoring? If so, suggest the appropriate abstraction(s), and optionally perform the refactoring. If you notice a repeating pattern, but decide it would not make sense to abstract it out, briefly explain why so.

(Hint: there are at least two patterns that appear more than once in the source code.)

e) *[very optional!]* Ideas! Is there anything else you would improve or change in the code?

(Aside from the distinct lack of a user interface; consider that a known feature.)

(Hint: such as: does it make sense to represent the wordlist as a class, as the rest of the program is a collection of functions? Which paradigm is better for this program, imperative or functional? If functional, are there sensible *opportunities for a more FP style?)*