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

Published 28.2.2018

Solutions for exercises, *lecture 3*

1. ★☆☆ *Imports.*

   a) The output is

   hello from a.py
   hello from b.py

   because in case of conflicting names, the latest star-import wins. “Latest” is considered dynamically, i.e. in the order the code is actually run.

   To place both imports at the top, and still call both versions of hello():

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

   This is also a valid solution:

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

   The main point here is that in Python, names belong to namespaces. To keep things explicit, we basically have two options. To refer to a name from another module, we may either:

   ● **import** the module that exports what we need, and then use the fully qualified (“dotted”) name, or:

   ● Import the name into the current scope as a bare name, using the **from**-import.

b) A **for** loop will do nicely:

```
from unicodedata import name
lst = ['ℝ', '数', 'ℓ', '⁂', 'あ', 'א', '⇨', '◊', '※', '♬']
for c in lst:
    print(c, name(c))
```

The output is:

```
ℝ DOUBLE-STRUCK CAPITAL R
数 CJK UNIFIED IDEOGRAPH-6570
ℓ SCRIPT SMALL L
⁂ ASTERISM
あ HIRAGANA LETTER A
א ALEF SYMBOL
⇨ RIGHTWARDS WHITE ARROW
◊ LOZENGE
※ REFERENCE MARK
♬ BEAMED SIXTEENTH NOTES
```

c) The point is that in Python, except for names defined in the current module and the language builtins, everything is imported from somewhere. For example, mathematical functions (those that Python's standard library itself provides) are contained in the *math* module.

※ *math* for numeric scalars. For numeric arrays, **import** numpy. For symbolics, **import** sympy.

Here is the **from**-import solution:

```
from math import sin, pi    # ← this line added

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

And this is the regular **import** solution:

```
import math    # ← this line added

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

d) Here is one solution for the color conversion:

```
import colorsys

def html_rgba_to_tuple(s):
    if not isinstance(s, str):
        raise TypeError("Expected str, got {}".format(type(s)))
    if s[0] != '#' or len(s) < 9:
        raise ValueError("'{:s}' is not an HTML color of the form '#rrggbbaa'".format(s))
    hex_strings = (s[k:k+2] for k in range(1, 9, 2))    # maybe over-engineered?
    numbers = (int(x, base=16) for x in hex_strings)
    r,g,b,a = (x / 255.0 for x in numbers)
    return r,g,b,a

s = '#6fa3f0ff'    # HTML RGBA, #rrggbbaa
r,g,b = html_rgba_to_tuple(s)[:-1]    # discard alpha
h,l,s = colorsys.rgb_to_hls(r, g, b)    # each component a float in [0, 1]
output_numbers = [int(255.0 * x) for x in (h, s, l)]    # each component an int in [0, 255]
output_s = '#{:02x}{:02x}{:02x}'.format(*output_numbers)    # hex string, but HSL
```

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

Data file (utf-8):

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

a) Loading and printing the file (ignoring exception handling in this simple exercise):

```
lines = []
with open('utf8_text_file.txt', mode='rt', encoding='utf-8') as f:
    for line in f:    # in Python, a text file behaves like a container for lines of text
        lines.append(line)
print(lines)
```

Running this script using python3 from the command line, it seems a gremlin has seeped in – some items contain a mysterious unprintable '\u3000'. What is a U+3000?

```
import unicodedata
unicodedata.name('\u3000')    # 'IDEOGRAPHIC SPACE'
```

...so likely the input method mode was wrong when typing some spaces into the example file when it was created!

b) Let's append, insert, replace and delete some lines, and save the result. Here's the complete script:

```
lines = []
with open('utf8_text_file.txt', mode='rt', encoding='utf-8') as f:
    for line in f:    # in Python, a text file behaves as an iterable containing lines of text
        lines.append(line)

lines.pop(-1)                               # remove the last line
lines.append("Hi, I'm a new line.\n")
lines.insert(0, "This line goes first.\n")
lines[1] = "Aáàäα... π ≈ 3.14159.\n"    # replace by writing into the list

# save the result
with open('output.txt', mode='wt', encoding='utf-8') as f:
    for line in lines:
        f.write(line)
```

Alternatively, we may use *print* with its *file* kwarg, and set the *end* kwarg to the empty string to suppress the automatic newline:

```
print(line, file=f, end='')
```

Yet another option, to write the file without an explicit **for** loop in our script, is to use *f.writelines(lines)*. The argument is an iterable, where each element is a line to write.

Note that if we use *write* or *writelines*, the lines need an explicit line terminator; unlike *print*, they do not add one automatically. On the other hand, if using *print*, we need to be careful not to introduce double newlines (per line, one from the input text, and one added by *print*).

c) Reading with a list comprehension:

```
with open('utf8_text_file.txt', mode='rt', encoding='utf-8') as f:
    lines = [line for line in f]
```

This is just two lines of code instead of four; which one is more readable, is a matter of opinion.

Note that since in Python, the smallest unit of scope is a function, we can simply create the *lines* name inside the **with** block; it will remain visible after the block exits.

If the read succeeds, the name will be created, and will point to a list instance that contains the text from the file. But if the read fails (e.g. file not found), the name will not be created!

In a real-world program, we would also use a **try**/**except** here to catch errors, using the pattern from lecture 3, slide 14.

3. ★★☆ *Language features.*

a) *LBYL vs. EAFP*

Take the LBYL mocks from the question paper. Here is the test program using the mocks:

```
def f():
    success = random_failure(0.5)
    if not success:
        return False
    success = random_failure(0.5)
    if not success:
        return False
    success = random_failure(0.5)
    if not success:
        return False
    return True

def g():
    func = DeterministicFailure(3)    # function factory, creates a function instance
    success = func()
    if not success:
        return False
    success = func()
    if not success:
        return False
    success = func()
    if not success:
        return False
    return True

def main():
    if not f():
        print("f failed")
    if not g():
        print("g failed")

main()
```

Note how, in the LBYL paradigm, each function call requires at least two lines of code; one to call the function, and another to check the return value.

(In “f” and “g”, we have used three lines, due to unfortunate naming: whereas the names of the mocks end in “*failure*”, what they actually return is whether the operation *succeeded*, as is the convention in practically all programming languages when a boolean is used as the return value.

Hence “*if not random_failure(0.5):*” sounds like the “if” is triggers if there was no failure, whereas it actually works in the exact opposite way. We name the return value “*success*” in order to explicitly document how to interpret it correctly.)

Next up, EAFP. Take the updated mocks from the question paper. Here is the test program:

```
def f():  # strategy 1: function that handles its own errors (no-throw guarantee)
    try:
        random_failure(0.5)
        random_failure(0.5)
        random_failure(0.5)
    except RuntimeError as e:
        print('f() failed, reason: {}'.format(e))    # show what happened
        # let's assume in this particular case it's ok to ignore the error
        # so we don't need to do anything more here

def g():  # strategy 2, also valid: function that lets errors propagate to caller
    func = DeterministicFailure(n=3, k=2)
    func()
    func()
    func()

def main():
    f()    # f handles its own exceptions, no need to catch anything
    try:  # g will raise if an error occurs, so catch any relevant exceptions here
        g()
    except RuntimeError as e:
        print('g() failed, reason: {}'.format(e))

main()
```

Error handling now takes up fewer lines of code, and the successful (usual) case has become more readable. The return value is now available to pass data, since it is no longer needed to pass error information. Furthermore, exceptions can easily carry a message indicating the reason of the error in plain text.

If the same handler is applicable to any error of some particular type, we only need to specify (or call) that handler once (not after every function call that may produce that kind of error).

If the programmer does not bother to catch errors, in LBYL the result is catastrophic: the operation will silently fail. This will likely corrupt the state of the program, because the later operations in effect assume that all earlier ones have been successful. The error will only be noticed later, after the context of the error has been lost, making the program hard to debug.

In EAFP, an uncaught exception that escapes to the top level will cause the program to terminate immediately. Hence, either the processing will stop (usually with an informative error message), or the programmer will implement an error handler to avoid that. More robust.

About the two strategies used in this test program, see Abrahams's exception safety guarantees.

However, keep in mind that EAFP is no silver bullet; even though it makes error handling appear simpler, it is still easy to write code that is *Cleaner, more elegant, and wrong*. (Any relevant errors must be handled *somehow*; no paradigm can remove the essential complexity.)

b) *Lexical vs. dynamic scoping.*

***Practical note**: if you keep the same Spyder session open and do various experiments in it, its global namespace will keep all the names you have assigned to at the top level.*

That behavior may interfere with this particular exercise, because an earlier definition of "x" may already have bound that name to some value.

Hence, it is useful to first open View ▷ Panes ▷ Variable Explorer, and click the × in its toolbar to reset the namespace (clearing all top-level definitions).

If the Variable Explorer is already open, it should appear in one of the panes at the right edge of the main window, usually as a tab in the same pane as Help. Click the tab header (at the lower edge of the pane) to switch to it, if necessary.

First of all, here are the **original** example programs from lecture 3, slide 26.

To place them in the same file (just for convenience) while avoiding name conflicts, here the functions have been renamed:

```
def f1():
    def g1():   # "g1" is defined inside "f1"
        print(x)
    x = 42
    g1()
f1()

def g2():
    print(x)
def f2():
    x = 42
    g2()
f2()
```

Running this script, we observe that "f1" indeed prints 42, while "f2" crashes with a NameError.

This is because lexically, there is no "x" in the scope of "g2" – i.e. within the piece of source code text spanned by the definition of "g2", or in one of its surrounding pieces of text (recall the "E" and "G" in **LEGB**).

In contrast, in "g1", there is an "x" in scope, since a surrounding piece of text – the definition of "f1" – defines an "x".

To try the dynamically scoped variant of these programs, we start by placing *dynscope.py* into the same folder with the test program.

Then, here is the code for the dynamically scoped version, using *dynscope.py* to emulate dynamic scoping:

```
from dynscope import env

def f1():
   def g1():   # “g1” is defined inside “f1”
      print(env.x)    # dynamic variables are accessed via env
   with env.let(x=42):
      g1()
f1()

def g2():
   print(env.x)
def f2():
   with env.let(x=42):
      g2()
f2()
```

Now both programs work; each program prints 42.

Even with the name “x” now dynamically scoped, also the first program still works, because the call to “g1” occurs within the *dynamic extent* of the **with** block, during which the name “x” exists.

[The ***dynamic extent*** of a block of code is the time during which that block is being executed.]

From the viewpoint of *dynamic* scoping, it is irrelevant whether “g1” is in the *lexical* scope of its caller (“f1”) or not. Dynamic scoping does not care *where* a function appears in the source code.

Finally, without add-on modules like *dynscope.py*, **Python always uses lexical scoping**.

This example is meant to illustrate the main difference between lexical and dynamic scoping, and that it is possible to emulate dynamic scoping in the rare case where having it available as an option can make code simpler.

It is still important to understand the concept of *dynamic extent*, because object instances – such as opened files – live dynamically (even if the names pointing to those instances are resolved lexically).

c) *Duck typing.*

About *callables* in Python: simple example, definition, and a built-in function to check if something is callable.

Consider the main routine from the question paper:

```
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)))
```

The **for** loop iterates over a tuple, which here is created inline and then discarded after the loop. (It is a temporary, only used for looping over.)

As for the items in the tuple:

● f is a function of one argument.

● g is a function of two arguments.

● Frizzle is a class that has been made callable by defining a *__call__* magic method.

Instances of Frizzle can be called as if they were functions; Python will then redirect the call to the *__call__* method of the object instance.

As always, the *self* argument is passed implicitly, so this *__call__* takes no arguments. Hence instances of Frizzle are callables of zero arguments.

Note that “Frizzle(23)” is not yet redirected to *__call__*; it creates an instance by calling the constructor, *__init__*. Recall that the syntax for creating an instance in Python is to *call the class*. The magic method *__call__* is invoked when we instead *call an object instance*.

Since *self* is passed implicitly, the positional argument 23 (that we pass to the constructor) goes into the first available formal parameter, namely “a”.

● Frobozz is also class that has been made callable. In this particular case, the constructor takes no arguments (except the implicit *self*), but *__call__* takes one argument, “a”. Hence instances of Frobozz are callables of one argument.

Again, Frobozz() just calls the constructor, and returns the created object instance.

● The class Counted is an example of where we might use a call-counting mechanism more sensibly. It takes a function, and returns a wrapped function that, in addition to performing its original task, counts how many calls have been made to it.

It is essentially a decorator (in the Python sense), although in the questions we used it directly, without any syntactic sugar.

Consider the implementation:

```
class Counted:
    def __init__(self, f):
        self.count = 0
        self.f = f
    def __call__(self, *args, **kwargs):
        self.count += 1
        return self.f(*args, **kwargs)
```

The constructor takes in an argument “f”, and stores it into *self.f*, so that instance methods (such as *__call__* here) can use it later. The programmer has clearly intended the argument to be a function, but being duck-typed, Python does not check that.

In a real-world program, the programmer would be expected to document this assumption in the docstring of the *Counted* class, or maybe verify it with **callable()**, and **raise** TypeError (with a descriptive error message) if it is not. Classes can be docstringed just like functions – if the first expression that appears in the class body is a string, it will be used as the docstring.

Now, what is the signature of the *__call__* method?

● *args means it takes any number of positional arguments, and packs them into a list, “args”.

● **kwargs does the same for keyword arguments, packing them into a dict named “kwargs”.

Hence, *__call__* accepts zero or more positional arguments, and any keyword arguments.
Its signature is completely generic – any arguments can be passed in, and it will accept them.
It can also be called with no arguments – in that case, “args” and “kwargs” will be empty.

As for the last line with **return**:

● *args unpacks the list “args” into the positional arguments of *self.f*.

● **kwargs, similarly, unpacks the dict “kwargs”, using its keys and values to pass arguments by name into *self.f*.

Hence, this particular *__call__ passes through any and all of its arguments to self.f*, i.e. to the function that the user passed in to the constructor.

In conclusion, which of the invocations succeed and which fail? The **try** block in main() expects a callable of one argument – so those invocations where *func* can be called with one argument succeed, while the rest fail. Explicitly, calls to *f*, *Frobozz()*, and *Counted(f)* succeed.

d) *Call-by-sharing.*

Questions, with answers:

- Is the *obj*, that each function receives, the same instance that is created in main()? (Deduce, or add appropriate prints and test.)

    - Generally yes, that is how call-by-sharing works. In this specific case, consider main() from the question paper:

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

    - The first line in main() creates an instance of “A”, and saves the reference as “o”.
    - This instance is then handed in to imperative_update(), by using the name “o”.
    - The same instance is handed in to functional_update(). The name “o” is then re-bound to the return value of functional_update().
    - Finally, this latest “o” is handed in to replace().

- Does replace() do what its name suggests? Why or why not?

    - No, it doesn't. The issue is that assigning to the name “obj” inside replace() only re-binds the name locally. Hence, after the assignment completes, “obj” in replace() will point to the string that was introduced there, whereas caller's “o” still points to the original object instance.

    In other words, the effect of the assignment is that replace() loses access to the object that was originally passed in as the formal parameter “obj”, because the assignment re-binds the only name by which replace() could access that object.

- Does imperative_update() work as expected? Why or why not?

    - Yes, it works as expected. It mutates the object instance that was passed in. The formal parameter “obj” points to the original object, so any modifications done to the object will be visible to the caller.

- What are the main differences between imperative_update() and functional_update()? In the definition? In the usage (i.e. at the call site)? Conceptually?

    - The key point in all three respects is that imperative_update() mutates the original object (and returns **None**), whereas functional_update() creates a new instance representing the new, “updated” state of the input, and returns the new instance. In functional_update(), the original object is not modified. The original object could as well be immutable, and it would still work.

    The most important implication of this is when considering the behavior of

complex pieces of software, where many parts of the program may have references to a specific object instance.

In the imperative approach, when some part of the code mutates the state, this changes the the original object, so the new state will immediately become visible everywhere. At first glance, this sounds convenient, but the approach has serious pitfalls – for example, rollback in error situations is difficult, and concurrent (in a sense "simultaneous") accesses may see the object in an inconsistent state, unless special precautions are taken (atomicity).

The functional approach makes it easier to solve these issues. First, we must realize that identity and state are two different things, which makes rollback trivial (simply do not commit the new state if something went wrong). It also implies that much less code needs to be exposed to concurrency issues.

As for the latter point, in terms of Python, consider having one central place that re-binds the name "world" to its new state – everywhere, data is immutable, and all reads receive a snapshot of the latest committed state. As long as this central update occurs atomically (so that if someone requests a snapshot while the update is in progress, they will either get the old state, or have to wait until the update completes, to get the new state), nothing else needs to be atomic as there is no way by which an inconsistent state could be produced.

Somewhat related to this is the vector clock (Fidge, 1988; Mattern, 1988), for imposing an ordering on partial state updates in a concurrent world, where each agent has access to only a snapshot of the others' states. (We will talk more about concurrency and parallelism later on this course.)

- In functional_update(), why is *type(obj)(**None**)* better than *A(**None**)*? *(Hint: OOP.)*

    - *type(obj)* always retrieves the actual type of *obj*, whereas hard-coding it as "A" makes the unnecessary assumption that the type will always be exactly "A".

      This assumption may fail much later, if the programmer derives a new class from "A" (let's call it "B"), adding some new functionality, or specializing it in some other way. If an instance of "B" is passed in to functional_update(), the hard-coded version will create an "A" for the return value, so information (any attributes specific to "B") may be lost, or behavior may be different from what the programmer expects from a "B".

- 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?)*

    - *type(A)* is *type*, because the built-in function **type** returns the type of its argument (which must be an object instance), and the name "A" is bound to a class – or in other words – to an instance of *type*!

      For backwards compatibility reasons, the name "type" refers to two different things:
        - The built-in function, "**type**", which retrieves the type of an object instance.
        - Python's default metaclass, "*type*". By default, **class**es are instances of *type*.

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

a) Implementing read-only properties that dynamically compute their values. Here is one possible way to do it:

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

    def get_perimeter(self):
        return self.a + self.b + self.c
    perimeter = property(get_perimeter, None, None, "The perimeter of the triangle.")

    def get_area(self):
        s = self.get_perimeter() / 2
        return (s * (s - self.a) * (s - self.b) * (s - self.c))**0.5
    area = property(get_area, None, None, "The area of the triangle.")

def main():
    T = Triangle(3, 4, 5)
    print(T.perimeter)    # 12, correct
    print(T.area)         # 6, correct (pythagorean triangle; base 3, height 4)
    T.a,T.b,T.c = 1, 1, 2**(1/2)
    print(T.perimeter)    # ≈3.414, correct (1 + 1 + sqrt(2))
    print(T.area)         # ≈0.5, correct (one half of the unit square)
main()
```

Alternatively, we could use the decorator syntax. Example in the documentation on built-ins. (Extra exercise: convert the above code to this variant.)

Note that the values of the perimeter and the area are never stored in the Triangle instance; they are re-computed each time they are requested.

b) Avoiding unnecessary re-computation. Here is the modified program:

```
class Triangle:
    def __init__(self, a, b, c):    # a, b, c: side lengths
        self.__a = a    # can't use set_a, because triggers calculations (__b, __c don't exist yet)
        self.__b = b
        self.__c = c
        self.update()

    def update(self):
        self.perimeter = self.calculate_perimeter()
        self.area = self.calculate_area()

    def calculate_perimeter(self):
        return self.__a + self.__b + self.__c

    def calculate_area(self):
        s = self.calculate_perimeter() / 2
        return (s * (s - self.__a) * (s - self.__b) * (s - self.__c))**0.5

    def get_a(self):
        return self.__a  # two underscores to make private (contrast protected)
    def set_a(self, x):
        self.__a = x
        self.update()
    a = property(get_a, set_a, None, "Length of side 'a'.")

    def get_b(self):
        return self.__b
    def set_b(self, x):
        self.__b = x
        self.update()
    b = property(get_b, set_b, None, "Length of side 'b'.")

    def get_c(self):
        return self.__c
    def set_c(self, x):
        self.__c = x
        self.update()
    c = property(get_c, set_c, None, "Length of side 'c'.")

def main():
    T = Triangle(3, 4, 5)
    print(T.perimeter)
    print(T.area)
    T.a,T.b,T.c = 1, 1, 2**(1/2)
    print(T.perimeter)
    print(T.area)
main()
```

c) In Python, *descriptors* are a mechanism to customize attribute access.

Instead of just a get, set or delete from the object's dictionary (*self*.__dict__), it is possible to make attribute accesses run custom code written by the programmer.

Properties are implemented in terms of descriptors; hence in this exercise, we have seen a small taste of what descriptors can do.

Perhaps the most important part of the descriptor howto (for the exercise to follow later) is:

… it is more common for a descriptor to be invoked automatically upon attribute access. For example, `obj.d` *looks up* `d` *in the dictionary of* `obj`*. If* `d` *defines the method* `__get__()`*, then* `d.__get__(obj)` *is invoked according to the precedence rules listed below.*

※ ***obj.d*** ⇒ ***d.__get__(obj)***; what's up with the swap in ordering? An example will help. Consider properties. The descriptor howto states that the built-in function **property** creates a data descriptor.

In item a), above, when we say:

perimeter = property(get_perimeter, None, None, "The perimeter of the triangle.")

what this technically means is that a descriptor object instance is created, and *that descriptor object instance* is bound to the name "perimeter". Because the binding occurs in the class scope, this assignment creates a method for the class Triangle.

(To be exact, about the assignment, we should only say *the name is bound in the class scope*. The right-hand side could as well be any value, in which case a class attribute would be created. Even now, it is indeed a class attribute that is created! In Python, an *instance method* is just a specific kind of *class attribute*; a callable, which takes in *self* as the first argument.)

Now, when the test program says:

```
T = Triangle(3, 4, 5)
print(T.perimeter)
```

Python notices, upon looking up the attribute T.perimeter, that it is a *descriptor object* that has the method *__get__*. For clarity, let's call this descriptor "D".

The access to T.perimeter is redirected to D.__get__(T) – i.e. the descriptor "D" is requested to return whatever it considers should be returned when one queries it in the context of the object instance T.

In a property, the *__get__* method is bound to the user-defined getter; in this particular case, *get_perimeter*. Hence, what will happen is that *get_perimeter* runs with *self*=T, and its return value becomes the result of the read access to T.perimeter.

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

a) The main ideas are that the *dynamic environment object itself, dynscope.env, is lexically scoped*, and that *data lives dynamically*.

Importing the module gains access to its top-level scope, so that the caller can access *env*.

By implementing a context manager, we can control what happens when a code block (that uses the **with** statement) is entered or exited – and this control occurs *dynamically*.

Hence, by storing the dynamically scoped names into the *env* object, and performing the appropriate magic in its *__enter__* and *__exit__* methods (see lecture notes, p. 40) to create and destroy these names, their visibility becomes determined by the dynamic extent of the **with** block – which is exactly how *dynamic scoping* behaves.

The last thing is to make sure only one *env* exists; this is done conveniently by making the implementation private to the *dynscope* module, and instantiating the object once, when the top level of the module runs. Since that occurs only once per session, this makes *env* a singleton (in the OOP sense).

b) Nested dynamic scopes are handled using a stack. The read and write operations are asymmetric in the sense that a read walks the stack in reverse order (innermost scope first) until the name is found, whereas writes always occur in the innermost scope.

c) The class *_EnvBlock* is the actual object being context-managed by **with**. It represents one level of dynamic scope.

The class *_Env* performs name lookup for the dynamic variables. It also provides *dynscope*'s only public method, *let*, which creates dynamic variables when used with **with**. This is done by instantiating and returning an *_EnvBlock*, on which **with** will then call *__enter__*.

d) In *_EnvBlock*, the *__enter__* method creates a dynamic scope by pushing the user-provided names and their values onto the stack, and the *__exit__* method deletes the currently innermost dynamic scope, by popping the last element off the stack.

To implement name lookup, the class *_Env* overrides attribute accesses, but using a mechanism we have not yet talked about. Whereas a descriptor is a high-priority way to override access for a single attribute, overriding the *__setattr__* and *__getattr__* methods provides a low-priority way to override ***all*** attribute accesses for the object. (Low priority here means that *__getattr__* ***will not be called***, if Python's regular attribute access succeeds.) Be careful when doing this.

The *__getattr__* override implements name lookup in the stack, starting from the innermost dynamic scope. When the using code performs a read access to *env.something*, what actually happens is that *env.__getattr__* runs with name='something', and its return value is returned as the result of the read access.

The *__setattr__* override makes sure that all dynamically scoped variables are managed by the intended mechanism, so that user code cannot directly write to the attributes of *env* (which would prevent *__getattr__* from working as intended).

e) A possible answer by just looking at the code:

For several modules using it: it should work, given how Python treats re-imports of the same module (i.e. by doing nothing except making the name visible at the import site).

Hence, the first import creates the *env* singleton, and gives access to it; whereas any later import simply gives access to the same singleton.

For several threads: the use of *threading.local* should ensure that each thread gets its own stack.

f) The implementation does **not** support re-binding of dynamically scoped names.

In the current implementation, the only way to introduce dynamically scoped names is via **with** env.let(x=…), and since *env.__setattr__* is overridden to raise an error, there is no way to modify their bindings. (This is actually the inspiration behind the no_rebind.py example, which demonstrates *__setattr__* and *__getattr__*.)

How could the feature be added? The obvious possibility is to modify *env.__setattr__*.

But first, stop – would such a feature even make sense? If it is required to change the binding of a dynamically scoped name *temporarily*, it is always possible to introduce another **with** env.let(x=…) using the new value – and because the implementation supports nested dynamic scopes, the *x* in the innermost dynamic scope will shadow any *x*'s in the outer ones.

If we really wanted to allow rebinding, here are some options:

● Make *__setattr__* write to the currently innermost dynamic scope (topmost item on the stack). This mimics how Python handles lexical scopes in assignment, in the absence of **nonlocal** or **global**.

● Make *__setattr__* search the stack like *__getattr__* already does, and if the name is found, overwrite with the new value. This is like **nonlocal**, but for dynamic scopes. Maybe dangerous for the readability of the code that uses *dynscope*. Keep in mind that the programmer has no lexical clues as to where the various levels of dynamic scope at a particular use site come from.

In either of these options, we may either:

● Allow only re-bind: require that the name already exists, raise an error if not.

● If not found, bind the name in the innermost dynamic scope. Maybe should not be allowed if using a **nonlocal**-like implementation, to avoid further damage to readability.

Which option feels most pythonic? Perhaps not providing re-binding at all. *Simple is better than complex*, and this also has an important readability advantage: *all* assignments to dynamically scoped names can be easily found in the source code by searching for *env.let* (or just *.let* to allow also for **as**-imports of *env* under some other name). Since it is rare to need this feature, that should produce only a few matches, viable for analysis manually.

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

a) Here is an overview. [The goal of this exercise was just to read about and roughly familiarize yourself with these components, not to explain them; in an answer, I wouldn't expect this kind of essay.]

- The program uses the *argparse* module of the standard library to provide a command-line interface, so that it can be invoked from a shell (command prompt). This is how it gets the filename of the source code file to analyze.

- The *logging* module is a flexible way to display messages in a command-line application or in a daemon (background process). The main advantages over print(…, file=sys.stderr) are control and convenience. The programmer can tag the *log level* of the messages as *error*, *warning*, *info* or *debug*. The logger can output to multiple destinations (handlers).

  Each destination may filter by the log level, showing only the messages considered important for that particular destination. For example, the same program can simultaneously produce a detailed debug log, while only printing errors and warnings to the terminal window – with no need to change the code at the call sites that send the messages to the logger.

- An abstract syntax tree (AST) is a representation of the abstract syntactic structure of the source code. ASTs are most commonly used by compilers, but they can also be used for automated source code analysis. Python provides the *ast* module, which gives access to the relevant part of the Python compiler – so that a Python program can take in Python source code, and from that generate the corresponding AST. This is a very powerful tool.

  Most programming languages, including Python, do not provide a way to convert an AST back to source code, but in case of Python someone has already thought of that – see astor (can be installed from PyPI).

  The Lisp family is an exception to this general rule. The surface syntax of Lisps is fairly close to the AST representation, which may at first seem weird, but it is an excellent choice if we want to process programs by programs – which is a central paradigm in the Lisp community. We will touch upon this in week 11, where we conclude our section on functional programming, and briefly discuss Lisps in general, and a modern Lisp, Racket.

- The visitor pattern separates the algorithm to process data contained in a data structure from the infrastructure needed to walk over the data structure. In this particular case, *ast.NodeVisitor* provides the infrastructure for walking over an AST, and invoking user-defined operations on specific Python AST node types. This particular program is only interested in *ClassDef* nodes, ignoring everything else. See also Green Tree Snakes, which advertises itself as *the missing Python AST documentation*.

b) Running the analyzer, the MRO of the test example is:

{'A': ['A'], 'D': ['D', 'B', 'C', 'A'], 'C': ['C', 'A'], 'B': ['B', 'A']}

This means that when calling a method of an instance of “A”, the method is looked up only in “A”. This is because “A” has no base classes (except the implicit *object*).

In the case of an instance of “D”, Python first looks in “D” itself. If the requested method is not found there, next up are “B”, then “C”, and finally “A”.

The detailed behavior of the algorithm can be observed from the log messages. Read the description of C3 linearization on Wikipedia, and compare the explanation to the log messages that the analyzer prints to the terminal window.

c) The example on Wikipedia, converted to Python, is:

```
class O: pass
class A(O): pass
class B(O): pass
class C(O): pass
class D(O): pass
class E(O): pass
class K1(A, B, C): pass
class K2(D, B, E): pass
class K3(D, A): pass
class Z(K1, K2, K3): pass
```

The MRO is:

{'E': ['E', 'O'], 'D': ['D', 'O'], 'A': ['A', 'O'], 'Z': ['Z', 'K1', 'K2', 'K3', 'D', 'A', 'B', 'C', 'E', 'O'], 'C': ['C', 'O'], 'B': ['B', 'O'], 'K3': ['K3', 'D', 'A'], 'K2': ['K2', 'D', 'B', 'E', 'O'], 'K1': ['K1', 'A', 'B', 'C', 'O'], 'O': ['O']}

The terminal output is now longer, because there are more classes to analyze, and the inheritance graph is more complex. The detailed behavior is of course very similar to the simpler case; it should be straightforward enough to follow the log messages.

d) It is possible to construct pathological examples, where the dependencies are specifically ordered in such a way that during linearization, the same class ends up both at the head of a list, and in at least one of the tails. For such cases, the algorithm finds no acceptable head, and hence no linearization exists (in the sense implicitly defined by the specification of the C3 algorithm).

Now knowing exactly what breaks the algorithm, let's construct such an example:

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

Feeding this to the analyzer, the analyzer exits with LinearizationImpossible.

(Here we have an example of a custom exception type, which has no implementation of its own – it reuses everything from its parent class, Exception. The reason we have made a custom type is that we can specifically catch it, without accidentally catching other exceptions.)

Note that a mere cycle in the inheritance graph does not preclude normal termination of the algorithm, provided that the implementation is shielded against infinite recursion by memoization of previous results.

For example, this example linearizes just fine, although it is **not valid Python**:

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

(Trying to actually run a script with these circular definitions, when we attempt to inherit “A” from “B”, or “B” from “C”, Python raises a NameError.)

At least our implementation (which should be correct, but no guarantees!) of the algorithm suggests that the MRO of this example should be:

{'B': ['B', 'C', 'A', 'B'], 'A': ['A', 'B'], 'C': ['C', 'A', 'B']}

which is nonsensical – in the MRO of “B”, “B” itself appears twice, and in the MRO of “A”, there is no “C”, although “B” inherits from “C”.

Academically speaking, how serious are these problems? A missed dependency certainly is.

How about the duplicate? For just looking up a method, it doesn't matter that “B” appears twice – if the method is already found in “B”, it is found at the first occurrence, and the rest of the list is never scanned. But if each implementation of the method (in “A”, “B” and “C”) calls super(), the program (if it was valid Python!) would enter an infinite loop – until the maximum depth of the call stack runs out.

On the other hand, perhaps the MRO of “B” is the most correct of the three, since it faithfully represents the cyclic dependency. Of course, in a real program these issues won't arise, since the inheritance graph (as a ***directed*** graph) will always be free of cycles.

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

a) How the sequence is generated:

- We use a recursive divide-and-conquer algorithm, which looks for sequences of positive integers that sum to *n*, the length of the input word. This problem is known as finding the compositions of *n*, or if ordering does not matter, the partitions of *n*. (Other, tangentially related problems are 3SUM, subset sum, and knapsack.)
- *k* is the number of letters in the current part. We iterate over $k = 1, 2, \ldots, n$.
- *m* is the number of letters remaining in the input word after *k* letters have been taken.
- Base case: if *m* is zero, the current part is the last one, and has *k* letters.
- Recursive case: use the same algorithm to split the remaining *m* letters into parts.
    - Because *m* is bounded ($1 \le m \le n$), and decreases at each step by at least 1, the algorithm is guaranteed to terminate.
    - The result is the part with *k* letters (the current one), plus the parts returned by the recursive invocation of the algorithm. Because there are in general many possible compositions, we iterate over the list of results.
    - If the recursive invocation yields no results, the problem was unsolvable – in that case, we yield nothing, as the result list is empty, so the loop over the results will not trigger. Here that will not happen, but we will use the same recursive divide-and-conquer pattern to solve another subproblem, where this observation (let's call it ⌂) will come in handy.
- The rest of the code in *split()* is a fail-fast check for the validity of the argument *n*, and a wrapper to invoke the generator – to make *split()*, which is a regular function, seem to the outside world as if it itself was the generator (that it actually wraps). This is the technique introduced in solutions 1–2, question 6.

How the sequence is transformed:

See the function *admissible_splits()*. The comments document the steps. Given the lectures so far, the code should be rather readable.

Let's analyze the most complex line:

```
splits = {the_split for the_split in splits if all(m >= min_component_length for m in the_split)}
```

- Set comprehension (curly braces), with a filter condition **if** …
- The built-in function **all** takes an iterable, and returns whether all of its elements satisfy a given condition. (A negative answer immediately short-circuits, skipping unnecessary further evaluation.)
- Arguments are evaluated before a function is called. Into **all**, we feed a generator expression, where the expression part is the truth value of (m >= min_component_length). The loop in the genexpr tells us that *m* iterates over the parts in *the_split*.
- **all** forces the generator, because it must evaluate the elements to compute the truth value.
- Hence, the filter condition accepts *the_split* only if all of its parts satisfy the minimum length condition.
- The loop in the set comprehension says that *the_split* iterates over *splits*.
- Finally, the expression part is just *the_split*, so the final result is to make a set of those splits that satisfy the filter condition.

Would this strategy be acceptable for very large *n*? No! Even if there were no other issues, the recursion will generate $n - 1$ nested function calls when it starts with the composition 1, 1, …, 1. This is unacceptable for unbounded *n*, because of the maximum size of the call stack. (See *sys.getrecursionlimit()*.)

But way before that starts becoming a problem, we run into the much more serious problem of computational complexity. Effectively, increasing the input word length by one will introduce another nested **for** loop, as the routine calls itself to solve the subproblem for length $n - 1$. The number of iterations taken by each loop is small, but still the complexity builds up very fast.

For illustration, let's use SimpleTimer from this example. (Exercise.) On my machine, split(20) takes ≈1.4 s to complete; split(21) already takes ≈2.8 s, split(22) ≈6.0 s, and split(23) ≈13 s. Each step approximately doubles the time needed for solving the problem of finding the compositions, so the benchmark indicates that time complexity is $O(2^n)$ – which is horrible.

However, it can be shown that there are 2^{n-1} compositions of $n \geq 1$ (proof in Wikipedia).
If our goal is to enumerate them all, *it is impossible to do better*, simply because we must add each one to the list of results, and doing this takes at least one operation per item.

Still, there is some **room for improvement**.

Memoization could help, by caching the solutions of already computed subproblems – this way, when the same number of remaining letters turns up again in another subproblem, we can re-use the already computed compositions. Here is a rough sketch of how this would work:

We split the function *split()* into two functions – an API wrapper that does the fail-fast check, and the actual computational wrapper (around the generator), which can be module-private. We only need the check when the user calls us; for any internal calls, we can arrange the preconditions to be always satisfied, so no check is needed for internal usage (likely faster).

The computational wrapper keeps a cache (memo) of results, keyed by the argument *n*.
To solve the subproblem, the loop calls the computational wrapper, not the generator directly.

Because a generator does not save the yielded results, in order to make the cache, we must force the generator, store the result in the cache, and then – if we want to keep the current interface to the outside world – return a generator that reads the results from the cache. This last part would be a genexpr, something along the lines of:

return (the_split **for** the_split **in** memo[n])

where memo[n] stores the compositions of *n*, so that the caller can still extract elements (individual compositions) at their leisure.

On the level of general principles, it is useful to filter as early as possible, as there is no point to waste time generating solutions that will be discarded later.

A simple concrete fix here would be to start the iteration of *k* from *minimum_component_length* instead of 1. The observation ⌂ above implies that the algorithm would automatically backtrack (discarding unsuccessful partial results) when the current attempt leads to a dead-end.

b) Let's analyze this:

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

- There is a generator expression inside the argument list to **all**. Arguments are evaluated first, before the function is called.
- In the genexpr, we iterate over the parts in *the_split*, and name the current part as *length*.
- The expression part of the genexpr is the result of the check *length **in** words_by_length*. Here *words_by_length* is a dictionary, keyed by word lengths. Thus, this checks whether at least one word of the given length exists in the input file (that contains words in the target human language which will be used for the parts of the anagram).
- Then, **all** forces the generator, and checks whether the condition holds for all items in the iterable.
- Finally, **not** inverts the result.

Hence, the code does as it pretty much says on the tin – it checks whether words of all required lengths (for this composition/split) exist in the input file.

This is a sanity check that enforces a precondition for the rest of the algorithm, so that later we may simply assume – without checking – that words of all required lengths do indeed exist. (Being able to make that assumption simplifies the rest of the code.)

c) The key is observation ⌂ from item a).

With this approach, *there is no need to implement explicit backtracking*; the call stack, and the state inside the function calls that have not yet terminated (strictly speaking, running generator instances that have not yet been exhausted), automatically keep track of the state for us.

d) The two patterns that spotted myself when writing the program were:

- The exact same divide-and-conquer recursive generator pattern shared by *_split()* and *comb()*.

  Perhaps no point in trying to extract this – the surrounding code can be very different in different use cases of the pattern, and *explicit is better than implicit*. Keeping the code as-is, there is one less abstraction the programmer needs to learn before being able to understand the rest of the source code.

  But it could be an interesting exercise to see how the extracted pattern would look, and whether the form of the result would suggest a succinct, self-explanatory name.

- To a lesser extent, the processing that is performed on the results in *admissible_splits()*, as well as in the main loop in *anagrams()*. Short; maybe no point in trying to extract this.

e) My own ideas were pretty much listed in the question: add a (command-line?) user interface; consider alternatives for implementing the wordlist. The problem feels like the FP paradigm is appropriate. Beside the wordlist, no immediate easy opportunities for a more FP style.