732a74-le2

February 10, 2019

1 Lecture 2: Functions and procedural abstraction, FP

- Functions
- Some brief notes on style, debugging and testing.
- Declarative patterns and functional programming.
- Presentation by Anders Märak Leffler
- Attribution: extends work by Johan Falkenjack.
- License: CC-BY-SA 4.0

2 Bonus information

- Early preview of Python 3.8 available. (Featuring walrus operator, apparently).

2.1 Why procedural abstraction?

In [ ]:
```
# A totally made up script, that goes a little overboard (for demonstration purposes).


# First we read the data
with ... :
    # some lines

# Then remove all the duplicates
# ...

# Then extract the relevant parameters by some algorithm.

# ... many lines of code cut and pasted ...
# ... many lines of code cut and pasted ...
# ... many lines of code cut and pasted ...
# ... many lines of code cut and pasted ...
# ... many lines of code cut and pasted ...


# Make sure that model_is_consistent is True iff the model is consistent (with some pr

# ... many lines of code cut and pasted ...
```

```
        # ... many lines of code cut and pasted ...


        if model_is_consistent:
            # Lots of code for plotting.
            # (without _any_ procedural abstraction this would be insane!)
```

```
In [1]: # Example, cleaned up.

        def run_simulation(fname):
            """Generate a model of data in file fname, and plot it if consistent."""
            data = read_data(fname)
            remove_duplicates(data)
            model = extracted_parameters(data)
            if model.is_consistent():
                plot(data, model)

        def read_data(fname):
            """Return the data contents of file named fname."""
            pass

        def remove_duplicates(dataset):
            """Remove duplicates from mutable input dataset."""
            pass

        def extracted_parameters(dataset):
            """Return a model based on the dataset, with features XYZ"""
            pass
```

- Structure and procedural abstraction.
  - Designing programs ("here I need *something that reads the data* and *something that cleans up the input*,..."), rather than "first I do instructions $s_0, \ldots, s_{1000}$ then jump back to step s_{349} if x is even...").
  - Readability.
  - Encapsulating behaviour.
  - Modularity in testing. Did everything fail because of a mistake in the "remove duplicates" chunk of code? We can test that more easily now.

- Practical scripting issues:
  - less repeated code (less "did I remember to fix the bug in *all* of the copy-pasted code?").
  - redo this calculation, but with other inputs.

- ... and much more ...

Caveats: * Sometimes having only a simple script makes sense. * Sometimes replacing function calls with their code makes *enough* sense from a performance perspective.

2.2 Functions in Python

- Typically defined with `def` or a `lambda`.
- First-class (callable) values. *A function is just an object.*
  - Corollary: you can't have two distinct f(x) and f(x,y) [without tricks].
- Called by `function_name(<arguments>)`.

```
In [1]: def hungarian_method():
            print("My hovercraft is full of eels.")

        def print_args(x):
            print("I was called with: ", x)
            return x
```

```
In [2]: hungarian_method()
```

```
My hovercraft is full of eels.
```

```
In [4]: print_args(99)
```

```
I was called with:  99
```

```
Out[4]: 99
```

```
In [5]: print_args(99) + 1000
```

```
I was called with:  99
```

```
Out[5]: 1099
```

```
In [7]: # What will this do? Crash? Print? Return?
        print_args(hungarian_method)
```

```
I was called with:  <function hungarian_method at 0x7fe97c4a21e0>
```

```
Out[7]: <function __main__.hungarian_method>
```

- Lambdas contain a single expression, and have implicit return.

```
In [11]: # Single argument lambda
         sq = lambda x : x**2

         # Two argument lambda
         add = lambda x, y : x+y

         # No argument lambda
         give_me_five = lambda : 5
```

```
In [13]: give_me_five()

Out[13]: 5

In [14]: bad_lambda = lambda x : while True:
             print("hello")


           File "<ipython-input-14-9ef3e7cd8ee8>", line 1
         bad_lambda = lambda x : while True:
                                 ^
     SyntaxError: invalid syntax
```

(Yes you can get around this, even without wrapping your code in a function, but you shouldn't. Use a `def` instead in that case.)

```
In [ ]: # Optional task: figure out how to make a lambda that imports the math module, request
        # prints the square root of that input when run.

        # This is really bad style, but might be fun to think about in order to understand the
        # This also suggests another reason as to why your code shouldn't eval(...) user input
        # suggest.
```

- When are arguments evaluated? Applicative-order evaluation.

```
In [15]: # What will this print?

         def f(n):
             print("--- Running f now!")
             return n

         print_args(print_args(f(100)))
```

```
--- Running f now!
I was called with:  100
I was called with:  100


Out[15]: 100
```

```
In [16]: # What happens to unused parameters?

         def test(use_2nd_arg, arg):
             if use_2nd_arg:
                 print(arg)
             else:
                 print("--- Didn't really need the value.")

         test(False, f(99))
```

```
--- Running f now!
--- Didn't really need the value.
```

Evaluates all arguments!

- Keyword arguments. Useful to clarify calls!

```
In [19]: test(arg=f(99), use_2nd_arg=False)
```

```
--- Running f now!
--- Didn't really need the value.
```

You can combine keyword arguments and regular positional, but positional come first.
[Note: Core developer Raymond Hettinger argues for (over)using this in PyCon 2013: Transforming Code into Beautiful, Idiomatic Python]

- Default arguments. Should follow non-default arguments.

```
In [22]: def test(use_2nd_arg, arg=5):
             if use_2nd_arg:
                 print(arg)
             else:
                 print("--- Didn't really need the value.")
```

```
In [24]: test(True)
```

```
5
```

```
In [26]: test(True, 99)
```

```
99
```

```
In [28]: test(arg=99)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

<ipython-input-28-d1b6e2c9da42> in <module>()
----> 1 test(arg=99)


TypeError: test() missing 1 required positional argument: 'use_2nd_arg'
```

Helps with understanding the documentation.

In [30]:
```
# What will happen here?

def add_to_list(e, lst=[]):
    lst.append(e)
    return lst

my_list = add_to_list(999)
add_to_list(123, my_list)
print(my_list)

my_other_list = add_to_list("bat")
add_to_list("man", my_other_list)
print(my_other_list)
```

```
[999, 123]
[999, 123, 'bat', 'man']
```

In [ ]:
```
# Think about later.

def add_to_list(e, lst=None):
    if lst is None:
        lst = []
    lst.append(e)
    return lst
```

- Functions can be defined within functions.
- Static/lexical scoping. What will the following print?

In [31]:
```
# What will this print?

x = 5
def f():
    print("I'm in f! Now x is ", x)

def g(x):
    print("I'm in g! Now x is ", x)
    f()

print("--- Calling f.")
f()

print("--- Calling g (which in turn calls f).")
g(99)
```

```
--- Calling f.
I'm in f! Now x is  5
```

```
--- Calling g (which in turn calls f).
I'm in g! Now x is  99
I'm in f! Now x is  5
```

What x was where f() was called from doesn't matter.

- LEGB lookup (first Local, then Enclosing, Global and finally Builtin scope).
- Binds values locally (unless told otherwise).

```
In [32]: # What will happen?

         x = 1
         print("Before, x is", x)
         def f():
             print("In f, x is", x)

         f()
         print("Afterwards, x is", x)
```

```
Before, x is 1
In f, x is 1
Afterwards, x is 1
```

```
In [33]: # What will happen?

         x = 1
         print("Before, x is", x)
         def f():
             x = 5
             print("In f, x is", x)

         f()
         print("Afterwards, x is", x)
```

```
Before, x is 1
In f, x is 5
Afterwards, x is 1
```

```
In [34]: # What will happen?

         x = 1
         print("Before, x is", x)
         def f():
             x = x + 1
             print("In f, x is", x)

         f()
         print("Afterwards, x is", x)
```

```
Before, x is 1
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)

<ipython-input-34-1c6c658e8533> in <module>()
      7     print("In f, x is", x)
      8
----> 9 f()
     10 print("Afterwards, x is", x)


<ipython-input-34-1c6c658e8533> in f()
      4 print("Before, x is", x)
      5 def f():
----> 6     x = x + 1
      7     print("In f, x is", x)
      8


UnboundLocalError: local variable 'x' referenced before assignment
```

```
In [ ]: # Exercise: play around with the keywords nonlocal and global.
```

- Function + their scope/context (the latter word used losely) form a **closure** (or "lexical closure").

```
In [37]: x = 1
         y = 2

         def f(x):
             def g():
                 print("In this g, x is", x, " and y is ", y)
             return g

         f(100)
         h100 = f(100)
         h55 = f(55)
```

```
In [39]: h55()
```

```
In this g, x is 55  and y is  2
```

```
In [40]: # Note: the lambda doesn't get x0 or x1 by parameter. It has to get them from somewhe
```

```
        def points_close_to(points, x0, y0, max_dist):
            is_near = lambda coord : (coord[0] - x0)**2 + (coord[1] - y0)**2 <= max_dist**2
            return list(filter(is_near, points))

        points = [ (1,2), (5,9), (3,3), (2, 5) ]
        points_close_to(points, 3,3, 3)
```

```
Out[40]: [(1, 2), (3, 3), (2, 5)]
```

```
In [170]: # Bonus exercise (way outside the scope of this course): can we reach into a closure
          # Python is flexible with certain things, and others not so much.

          h100 = f(100)
          print(h100)
          # Play around with h100. Check the dunder methods. (those that start and end with __
```

```
<function f.<locals>.g at 0x7f1758418598>
```

3 A note on naming, style, testing

- You should understand your code.
- Others should understand your code (and convince themselves of its correctness).
- . . . one of those 'other people' is future you.

3.0.1 Python function style interlude

> Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

The "Overriding Principle" from the official PEP 8 – Style Guide for Python Code.
In other words: focus on *what* it does (or return), not *how* it does it.

```
In [45]: # Slightly obfuscated. What's a good name?

        def f(x):
            for y in x:
                x[y] += 1
```

```
In [44]: seq = [1,2,3]
         f(seq)
```

```
---------------------------------------------------------------------------

IndexError                                Traceback (most recent call last)

<ipython-input-44-ad28de7c271a> in <module>()
  1 seq = [1,2,3]
```

```
----> 2 f(seq)


<ipython-input-43-177c42108ed8> in f(x)
  3 def f(x):
  4     for y in x:
----> 5         x[y] += 1


IndexError: list index out of range
```

```
In [47]: d = {"a" : 0, "b": 1}
         f(d)
         d

Out[47]: {'a': 1, 'b': 2}

In [85]: def increment_vals(mapping):
             """Increase all values in dictionary mapping by one."""
             for key in mapping:
                 mapping[key] += 1      # note that mapping[key] += uses a method of the mappi


         help(increment_vals)
```

```
Help on function increment_vals in module __main__:

increment_vals(mapping)
    Increase all values in dictionary mapping by one.
```

```
In [86]: # Clarification after lecture: ad hoc code to demonstrate modifications within an obj
         # (As opposed to changing what a certain name means locally, see result = ... example

         def square(seq):
             for i, val in enumerate(seq):
                 #seq[i] = val*val   # change something at place i
                 seq.append(val*val)   # even clearer extension.

         # rule of thumb: brackets and = or += likely changes

In [52]: increment_vals.__doc__

Out[52]: 'Increase all values in dictionary mapping by one.'
```

Consider * function name. * parameter names. * variables.

```
In [41]: help(max)
```

```
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

Not required in labs, but useful. Official guidelines: PEP 257 – Docstring Conventions.

3.1 A few general hints for design and naming

Caveat: large subject, involving personal preferences and official guidelines (sometimes at company level).

- Struggling with naming can mean that a function does too many different things.
  - Separate effects (print this, write file, change data by deaveraging...) from computation (is this valid data? What is the mean of this?).
- Take the **calling function's perspective** when naming.
  - Ex: When writing a consistency-checking function, consider if `if is_consistent(data) ...` would more sense than `if calculate_if_data_consistent(data) ...` *for the function calling the one you're writing.*
- **Don't mislead**. `is_consistent(data)` should test if it is consistent, not change the data, print something or the like.
- **Consider the application!** What does the name represent? (Cf last lecture.)

```
In [53]: # Example
         names = ["Alonzo", "Zeno", "Trisse"]

         # Bad. j sounds like an index. (i,j,k...)
         for i, j in enumerate(names):
             print("The {}th name is {}.".format(i,j))
```

```
The 0th name is Alonzo.
The 1th name is Zeno.
The 2th name is Trisse.
```

```
In [54]: # OK, we know that it's a list of strings. But the strings _represent_ something.
         for i, string in enumerate(names):
             print("The {}th name is {}.".format(i,string))
```

```
The 0th name is Alonzo.
The 1th name is Zeno.
The 2th name is Trisse.
```

```
In [55]: for i, name in enumerate(names):
             print("The {}th name is {}.".format(i,name))
```

```
The 0th name is Alonzo.
The 1th name is Zeno.
The 2th name is Trisse.
```

- In *small-scale* top-down design, creating "empty" functions with a certain contract can be helpful.
  - Example: "In my program I will need a function `read_data` which takes a file name for a CSV file and returns a data set on format *X*." (this is the contract) "I'll define it, and fill it in later."
  - A Python function body cannot be empty. `pass` is your friend.

```
In [ ]: def read_data(fname):
            pass  # might return some useful test data first, and then have
                  #file reading capabilities.

        # continue coding.
```

3.2 A note on testing and (unsystematic) debugging

> It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise - this thing gives out and [it is] then that 'Bugs' - as such little faults and difficulties are called - show themselves and months of intense watching, study and labor are requisite.

Thomas Edison, quoted in Ammann & Offutt (2017) - Intro to Software Testing, 2nd ed.

- Plenty of tools out there for debugging.
  - One example: pdb.
- We will focus on simpler print-style and asserts.

```
In [ ]: # Faulty code. Somewhat artificial example.
        # Also an inefficient method, but let's not worry about that.

        def calculate_if_prime_or_not(x):
            # Check to see if we can find something that divides the number.
            for y in range(x):
                if y % x == 0:
                    answer = False
                else:
                    answer = True
            return answer
```

```
In [2]: def is_prime(x):
            # Check to see if we can find something that divides the number.

            answer = True
            for y in range(2,x):
                #print("is", x, "div by", y)
                if x % y == 0:
                    answer = False
                    return False

                #print(answer)
            return answer

        # Print values.
        # What is the expected state? Should False change to True? True to False?...
        # Errors. Runtime. (not nice way)
        is_prime(53)

Out[2]: True

In [72]: is_prime(4)

is 4 div by 1
is 4 div by 2
is 4 div by 3


Out[72]: True
```

What are the issues? Found how?

```
In [222]: # What is expected?
          # Printing!
          #...

Out[222]: True

In [1]: def update_result(result, i):
            # print("update result got: ", result, i)
            result = result + i   # Merely changes the label "result".
            # With brackets, .append or the like => probably changes the
            # state of the object.

            # This only sticks the local label "result" on old result + i

        def sum_values(n):
            result = 0
            for i in range(n):
                print("Before: ", result)
```

```
        #         update_result(result, i)   # Original line from the code.
                result = result + i
                print("After: ", result)   # Strategy: we printed before and expected this to
                print()
            return result

        print(sum_values(5))
```

```
Before:  0
After:  0

Before:  0
After:  1

Before:  1
After:  3

Before:  3
After:  6

Before:  6
After:  10

10
```

In [3]:
```
        # Bonus segment: an obvious programming way to make it faster (without square roots et

        # Declarative,
        def is_prime_length_based(n):
            return len([d for d in range(2,n) if n % d == 0]) == 0
            # could be written not len(...)

        def is_prime_breaking(n):
            return None not in (None for d in range(2,n) if not n % d)

        # What happens if we change () into []?

        import profile
        N = 99999999
        print("--- is_prime")
        profile.run("is_prime({})".format(N))

        print("--- is_prime_breaking")
        profile.run("is_prime_breaking({})".format(N))

        print("--- is_prime_length_based")
        profile.run("is_prime_length_based({})".format(N))
```

```
--- is_prime
         5 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(exec)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 <ipython-input-2-140e1278863b>:1(is_prime)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 profile:0(is_prime(99999999))
        0    0.000             0.000          profile:0(profiler)


--- is_prime_breaking
         7 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(exec)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 <ipython-input-3-ee01cef06ea4>:8(is_prime_breaki
        2    0.000    0.000    0.000    0.000 <ipython-input-3-ee01cef06ea4>:9(<genexpr>)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 profile:0(is_prime_breaking(99999999))
        0    0.000             0.000          profile:0(profiler)


--- is_prime_length_based
         7 function calls in 6.200 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    6.200    6.200 :0(exec)
        1    0.000    0.000    0.000    0.000 :0(len)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    6.200    6.200 <ipython-input-3-ee01cef06ea4>:4(is_prime_length_
        1    6.200    6.200    6.200    6.200 <ipython-input-3-ee01cef06ea4>:5(<listcomp>)
        1    0.000    0.000    6.200    6.200 <string>:1(<module>)
        1    0.000    0.000    6.200    6.200 profile:0(is_prime_length_based(99999999))
        0    0.000             0.000          profile:0(profiler)
```

- We can add assertions to show that our (pure) function passed some useful tests. Corner

cases to illustrate?

```
In [92]: # assert is_prime(???) == ???, "???"
         assert True, "this should succeed"
         #assert False, "this should fail"
         assert not is_prime(4), "known non-prime 4 should not be considered prime"
         assert is_prime(2), "the even number 2 should be considered prime"
         #assert ...

         # assert statement

         # Add corner cases. 2 is the only non-odd prime. Empty lists...
         # Cases both for expected success and failure.

         print("All tests OK.")
```

```
All tests OK.
```

[Note: We might have other forms of properties that might be suitable for automatic testing. For instance "all return values should be True/False", "all multiples of two integers should be non-prime" or the like. This is outside the scope of this course, but an interesting subject to look into.

Modest focus here: strategies to find good test cases to convince ourselves somewhat.]

- Some of these can be expressed as input-output pairs. How to use?

```
In [98]: def test_is_prime():
             # tuple of test cases
             tests = (
              (4, False, "known non-prime 4 should not be considered prime"),
              (2, True, "the even number 2 should be considered prime"))

             for inputs, expected_output, error_msg in tests:
                 assert is_prime(inputs) == expected_output, error_msg

             print("--- test_is_prime succeeded")
         test_is_prime()
```

```
--- test_is_prime succeeded
```

3.3 New type interlude: namedtuples

- [Note and heading added after lecture, for easy reference.] Behave like tuples, but also allow member access by names. Eg mytuple.args instead of mytuple[0]. Enhanced readability.

```
In [104]: from collections import namedtuple
          test_case = namedtuple("testcases", ["args", "expected_output", "error_msg"])
```

```
        def test_is_prime():
            # tuple of test cases
            tests = (
             test_case(4, False, "known non-prime 4 should not be considered prime"),
             test_case(2, True, "the even number 2 should be considered prime"))

            for case in tests:
                assert is_prime(case.args) == case.expected_output, case.error_msg

            print("--- test_is_prime succeeded")
        test_is_prime()
```

```
--- test_is_prime succeeded
```

[Note: we can test functions which depend on state as well. We need to set up (relevant) state, run tests and tear it down afterwards. You might want to look at eg unittest or other frameworks if you are interested in this.]

4 Paradigms

- Philosophies about code.
  - Execution model.
  - Code organization.
- Python is a multi-paradigm language (sort of).

5 Functional programming

- Typically declarative.
- Purity (see below) makes it easy to reason about. (Not necessarily write...)
  - In particular for parallel computation.
  - MapReduce.

5.0.1 Pure functions

- No side causes.
  - Doesn't depend on the state (of the world).

```
In [123]: import datetime

        def has_side_causes():
            return datetime.datetime.now()

        def no_side_causes():
```

```
            return 5 + 5

        has_side_causes()
```

```
Out[123]: datetime.datetime(2019, 2, 8, 14, 52, 2, 520047)
```

- No side effects.
  - Doesn't change the state of the world.

Corollary: In (purely) functional programming, * there will be less of hidden dependencies on state. * reduced complexity. * referential transparency. Calling function f with input X will always yield the same result. * mathematical resoning useful (not just "do X, then Y").

5.1 Feature: higher order functions

- Functions with functions as outputs are generally considered higher-order. Have you seen those before? (Functions + scope.)

```
In [2]: # Exercise.
```

- Similarly with functions taking functions as input, or returning functions as output.

```
In [202]: # Implementing our own filter, with a list comprehension.

          def points_satisfying(points, to_keep):  # to_keep behaves as a function
              """Return a list of points which satisfy the criterion set out by the to_keep fu
              return [p for p in points if to_keep(p)]  # if we treat it like one

          points = [ (1,2), (-2,3), (5,100) , (0,-1) ]
          points_satisfying(points, lambda p : p[0] >= 0 and p[1] >= 0)
```

```
Out[202]: [(1, 2), (5, 100)]
```

5.2 Declarative patterns

- Many declarative-style patterns are Pythonic.
- Often implemented with iterators.
- Useful to chain together, one result feeding into the next.
- Often well-implemented.
- Example: map.

```
In [124]: # map is a standard functional programming pattern.
          # Cf mathematical mapping. map takes a sequence of values and produces the image of

          print(map(lambda x : x**2, [1,2,3]))
          tuple(map(lambda x : x**2, [1,2,3]))
```

```
<map object at 0x7fe97c406908>
```

```
Out[124]: (1, 4, 9)
```

```
In [125]: # Caveat: in Python this produces something which generates the values as you ask fo

          squares = map(lambda x : x**2, [1,2,3])
          print(squares)
          print(list(squares))   # Consumes all the values.
          print(tuple(squares))  # All consumed.
```

```
<map object at 0x7fe97c3f2d30>
[1, 4, 9]
()
```

Corollary: this can be quite efficient. Doesn't generate values in vain.

- Example: filter

```
In [126]: # filter keeps those values which conform to a predicate (function which - ideally -
          # on if a criterion has been satisfied).

          is_even = lambda n : n % 2 == 0       # could be written "not n % 2"
          filter(is_even, [1,2,3])
          list(filter(is_even, [1,2,3]))
```

```
Out[126]: [2]
```

- Example reduce.

```
In [232]: from functools import reduce

          reduce(lambda acc, e : [acc] + [e], [1,2,3, 4])   # [  [ [1,2] ] + [3]  ] + [4]
```

```
Out[232]: [[[1, 2], 3], 4]
```

- These can be combined.

```
In [129]: sq = lambda x : x**2
          seq = range(100)

          square_evens = map(sq, filter(lambda n : n % 2 == 0, seq))
          #list(square_evens)
```

```
Out[129]: [0,
           4,
           16,
           36,
           64,
           100,
           144,
```

```
196,
256,
324,
400,
484,
576,
676,
784,
900,
1024,
1156,
1296,
1444,
1600,
1764,
1936,
2116,
2304,
2500,
2704,
2916,
3136,
3364,
3600,
3844,
4096,
4356,
4624,
4900,
5184,
5476,
5776,
6084,
6400,
6724,
7056,
7396,
7744,
8100,
8464,
8836,
9216,
9604]
```

5.3 A note on recursion

- Practiced in lab 2B.
- Function values defined in terms… of function values.

- Focus on mathematical identities.
- Sometimes useful in its own right.
- Sometimes inefficient, but useful when turned into non-recursive algorithms.
- Use cases here: many tree-structures.

- Strategy
  - "Pretend" that we know the solution to a smaller case, extend that.
  - What do we know about the *smallest* case (or the smallest cases)?

```
In [4]: # What is the sum 0 + 1 + ... + (n-1) + n?

        # Helper function (during program construction).
        def sum_reference(n):
            """Return the sum 0+...+n."""
            return sum(range(n+1))

        # If we knew the sum 0 + 1 + ... + (n-1)?

        def sum_rec(n):
            pass
```

- Handwaving explanation above. Good mathematical foundations.
  - Closely related to inductive proofs.
  - Well-orderings.

- Sanity checks:
  - base case (or base cases). No more subdivision of the problem. Typical examples: empty list, zero, reached the leaves in a decision tree.
  - recursive case. Split into smaller case, and write the solution to this case. (Typically: rest of the list, traverse left or/and right branches of the tree...)