Module 9: Recursion and Fractals

Exercise

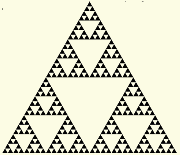
If you have not already, get prepared for class by downloading the start code:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/module-09-start.ipynb
```

Simply put, recursion is any thing that refers to itself.

Some examples:

- This sentence is recursive, since it is talks about itself.
- Recursive acronyms: “GNU’s Not Unix!”, and the mutually-recursive pair, “Hird of Unix-Replacing Daemons”, “Hurd of Interfaces Representing Depth.”
- Many fractals, including the Sierpiński Triangle:



Recursion is an important concept; we need a deep understanding of what it can do.

There are some nice examples on Wikipedia.

Fractals

One of the nicest place to see recursion is in **fractals.** A fractal is a thing that has self-similar “copies” of itself in itself.

https://en.wikipedia.org/wiki/Fractal

Canvas

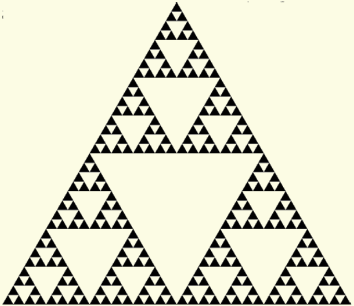
We want to draw fractals. So we need a drawing tool.

We will use the `ipycanvas` module. It gives us a “canvas” that we can paint on.

See the demo in the module start code.

A Recursive Fractal: the Sierpiński triangle

This is an image of the fractal we are aiming to create:

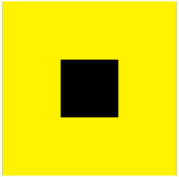


To see a detailed walkthrough of creating it, run the following to get a notebook:

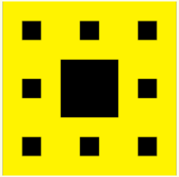
```
!wget https://student.cs.uwaterloo.ca/~cs114/src/sierpinski.ipynb
```

(Run this line from the starter code.)

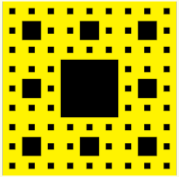
A **Sierpiński carpet** is a square fractal that is divided into 9 smaller squares: the middle is simply filled in, and each of the outer 8 squares is a smaller Sierpiński carpet, of depth one less. Like so:



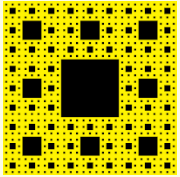
depth=0



depth=1



depth=2



depth=3

Exercise

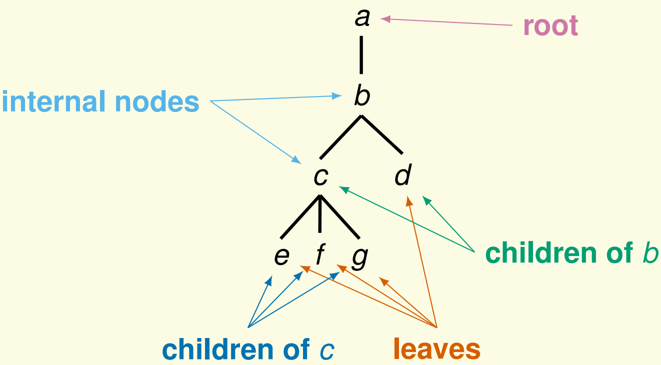
Look at the starter code.
Complete the function `carpet(canvas, x0, y0, width, depth)` so it draws a carpet like those above.

A Recursive Data Structure: Trees

There are many situations where we need to a data type that contains items of the same kind. Here are just a few examples:

- In computers, a **directory** may contain other directories, which may in turn contain other directories.
- A person may have **descendants**, who are themselves people, and may in turn have descendants.
- In linguistics, a **clause** may contain other clauses, and may in turn contain clauses.
- There are substantial applications in astronomy (https://arxiv.org/abs/0801.2004) and particle physics (https://arxiv.org/abs/1608.04772).

Tree Terminology



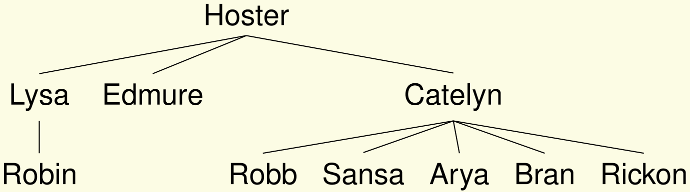
c is a **sibling** of d, and d is a sibling of c. e, f, and g are siblings of each other.

a is a **parent** of b. b is a parent of c and d. c is a parent of e, f, and g.

A node with no children is a **leaf**.

Family Tree

Consider a tree representing a **family**: a **person** and all their descendants:



For clarity, let's use a **class.**

It needs to store the person's **name**,
and a representation of the **family** of each of their children.

```
class Family:
    """Store information about the Family of a person."""
    name: str
    children: list['Family']
```

This class does not need to do anything tricky. It literally just stores the `name` as a `str`, and the `children` as a `list[Family]`. (Add a `__repr__` function to make it pretty.)

Here is all we need:

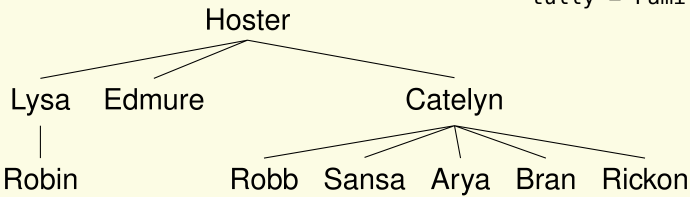
```
class Family:
    """Store information about the Family of a person."""
    name: str
    children: list['Family']

    def __init__(self, name: str, children: list['Family']) -> None:
        self.name = name
        self.children = children

    def __repr__(self) -> str:
        return "Family('" + self.name  + "', " + str(self.children) + ")"
```

`Family` class

To create a `Family`, we call `Family` with 2 args: a **`str`** for `name`, and a **`list`**`[Family]` for `children`.




```
tully = Family('Hoster',
               [Family('Lysa',
                       [Family('Robin', [])]),
                Family('Edmure', []),
                Family('Catelyn',
                       [Family('Robb', []),
                        Family('Sansa', []),
                        Family('Arya', []),
                        Family('Bran', []),
                        Family('Rickon', [])])
               ])
```

Look carefully; notice for example that Robin has 0 children, so an empty list; Lysa has one child, so that list contains only one `Family`.

Here a `Family` is a “leaf” if it has no children.
So in this example the leaves are Robin, Edmure, Robb, Sansa, Arya, Bran, Rickon.

Example: Counting people in a `Family`

To count the number of people in a `Family`, there are 2 kinds of people to consider:

- The person whose name is `name`, and
- All the people in all the `Family` values in `children`.

Let's carefully think about how to count how many people are in a `Family`.

```
def count_members(fam: Family) -> int:
    """Return the number of people in fam."""
    total = 0
    for child in fam.children:
        ## fam.children is a list[Family], so child is a Family.
        total = total + count_members(child)
    ## ...and one more for the person whose name is fam.name.
    total = total + 1
    return total
```

And sure enough it works:

```
check.expect('count tully', count_members(tully), 10)
```

Practice: Flattening a tree

```
def count_members(fam: Family) -> int:
    """Return the number of people in fam."""
    total = 0
    for child in fam.children:
        ## fam.children is a list[Family], so child is a Family.
        total = total + count_members(child)
    ## ...and one more for the person whose name is fam.name.
    total = total + 1
    return total
```

Exercise

Using this as a model, write a function `list_names(fam: Family) -> list[str]`.
It returns a `list[str]` containing all the names of everyone in `fam`, in alphabetic order.

```
list_names(tully)
⇒ ['Arya', 'Bran', 'Catelyn', 'Edmure', 'Hoster', 'Lysa', 'Rickon', 'Robb',
   'Robin', 'Sansa']
```

Hint

Collect the names in `answer`, then use `return sorted(answer)` so they're in order.

Working with Leaves

Recall that a "leaf" is a node that has no children.

In our representation, if `f` is a `Family`, it is a leaf if `f.children == []`.

Exercise

Write a function `family_leaves` that takes a `Family` and returns the number of "leaves", that is, how many people with 0 children.

```
check.expect("CLft", family_leaves(tully), 7)
```

Hint

Consider: what must `family_leaves(Family('Robin', []))` return?

Exercise

Write a function `leaf_names` that takes a `Family` and returns a `list[str]` containing the names of all the leaves, in alphabetic order.

```
leaf_names(tully) ⇒ ['Arya', 'Bran', 'Edmure', 'Rickon', 'Robb', 'Robin', 'Sansa']
```

Leaf-Labelled Trees

In the `Family` class, the `name` attribute stores some information on each node. We call it a "label".

A **leaf-labelled tree** is a special kind of tree where we have labels only on the **leaves**.

If we don't have the label, a node is storing only a list of... more trees. In that case we can do without the `class`, and just use a `list`. We call the result a **leaf-labelled tree**.

Let's just use a `int` as the label of each leaf. We will define an `LLT`:

> an `LLT` is either a `int`, or a `list[LLT]`.

In Python we can write this as:

```
LLT = int | list['LLT']
```

The "`|`", which we call a "pipe", means something like "or".

(We should also say that it is not empty.)

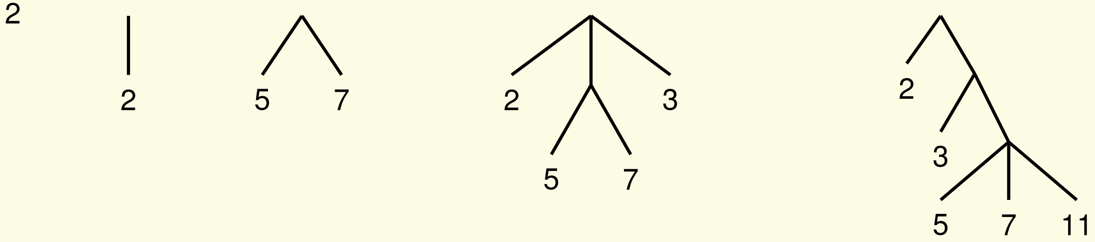
Here are some leaf-labelled trees as code, and as diagrams.

Consider how each satisfies the definition:

```
LLT = int | list['LLT']
```

...and also how the diagram corresponds to the code.

`2` `[2]` `[5, 7]` `[2, [5, 7], 3]` `[2, [3, [5, 7, 11]]]`



In the first example, the root **is** a leaf.

In the second example, the root has one child; that child is a leaf.

Distinguish node types

Recall the definition of a LLT:

```
LLT = int | list['LLT']
```

In code, we will usually need to treat differently the different “kinds” of LLT. We will write:

```
if isinstance(t, int): ...
if isinstance(t, list): ...
```

Then we can treat leaf nodes differently from non-leaf nodes.

Example: Counting leaves in a LLT

Keeping in mind that a LLT is either an `int` or a `list[LLT]`, we want to complete this function:

```
def count_leaves(t: LLT) -> int:
    """Count how many leaves are in t."""
```

- If `t` is an `int`, how many leaves are there? 1. That's our **base case**. We can write the code:

  ```
      if isinstance(t, int):
          return 1
  ```

- If `t` is not an `int`, it's a `list[LLT]`.
  Each item in `t` is a LLT; we want to know the total number of leaves.
  Determine how many are in each child; add them up.

  ```
      elif isinstance(t, list):    # (We could just use `else`.)
          total = 0
          for child in t:
              total = total + count_leaves(child)
          return total
  ```

It works: `count_leaves([2,3, [4], [2,[6,7]]])` ⇒ 6

Practice with LLT

```python
def count_leaves(t: LLT) -> int:
    """Count how many leaves are in t."""
    if isinstance(t, int):
        return 1
    elif isinstance(t, list):    # (We could just use `else`.)
        total = 0
        for child in t:
            total = total + count_leaves(child)
        return total
```

Exercise

Using this code as a model, write a function `sum_leaves(t: LLT) -> int` that takes a LLT and returns the **sum** of all the labels. For example,

`sum_leaves([2, 3, [4], [2, [6, 7]]])` ⇒ `24`

Consider the same questions:

- What to do if it's an `int`?
- What to do if it's a `list[LLT]`?

So far we've only used recursion to work with tree-like things.

Recursion is particularly useful with trees, but we can also use it to do other computations.

In fact, it's in principle possible to do **any** calculation without loops, using only recursion.

Let's see how.

The factorial function, written $n!$, takes a natural number and returns the product of numbers from n down to 1. For example, $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

But $4! = 4 \times 3 \times 2 \times 1$. So notice that $5! = 5 \times (4 \times 3 \times 2 \times 1) = 5 \times 4!$.

We can generalize this, and define this function recursively using mathematical language:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

When n is 1, we don't need to do any calculation; the answer is just 1. This is the **base case.**

Otherwise, we do a calculation that includes a call to the same function with different parameters (specifically, n is smaller by 1). This is the recursive case.

Implementing Loops with Recursion: example: factorial

We take this mathematical definition:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n \times (n-1)! & \text{otherwise} \end{cases}$$

... and directly translate it into Python code:

```python
def factorial(n: int) -> int:
    """Return n!"""
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Compare this to a version written using a `while` loop:

```python
def factorial(n: int) -> int:
    """Return n!"""
    product = 1
    while n != 1:
        product = n * product
        n = n - 1

    return product
```

- Both stop when `n == 1`
- Both move `n` closer to the base by "replacing" `n` with `n - 1`.

We can always do something like this; any code written with a loop we can rewrite to use only recursion.

Example: countdown

I can read this code to mean "to calculate $n!$, find $(n-1)!$ and multiply by n."

```
def factorial(n: int) -> int:
    """Return n!"""
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

In a similar style, I can say: "to count down from n to zero, print n, and then count down from $n-1$ to zero."

Exercise

Without a `for` or `while` loop, write a function `countdown_rec(n: int)` that prints all the numbers from `n` down to `1`. For example, `countdown_rec(5)` should print 5, 4, 3, 2, 1, on separate lines.

That is, rewrite this `countdown` without a loop:

```
def countdown(n: int) -> None:
    while n != 0:
        print(n)
        n = n - 1
```

Recall: you can create a list that contains the contents of two other lists, using `+`.

For example, `[2, 4] + [6, 0, 1]` ⇒ `[2, 4, 6, 0, 1]`

Exercise

Use this to write a function `countdown_list` much like `countdown`, that returns a `list[int]`, instead of printing them.

`countdown_list(4)` ⇒ `[4, 3, 2, 1]`

`countdown_list(5)` ⇒ `[5, 4, 3, 2, 1]`

Use the fact that `countdown_list(5) = [5] + countdown_list(4)`.

Example: the Fibonacci sequence

The **Fibonacci sequence** is a sequence of numbers that can be define as follows:

$$F(0) = 0; \qquad F(1) = 1; \qquad F(n) = F(n-1) + F(n-2) \text{ for } n > 1$$

We get a sequence of numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

We can easily implement this in code:

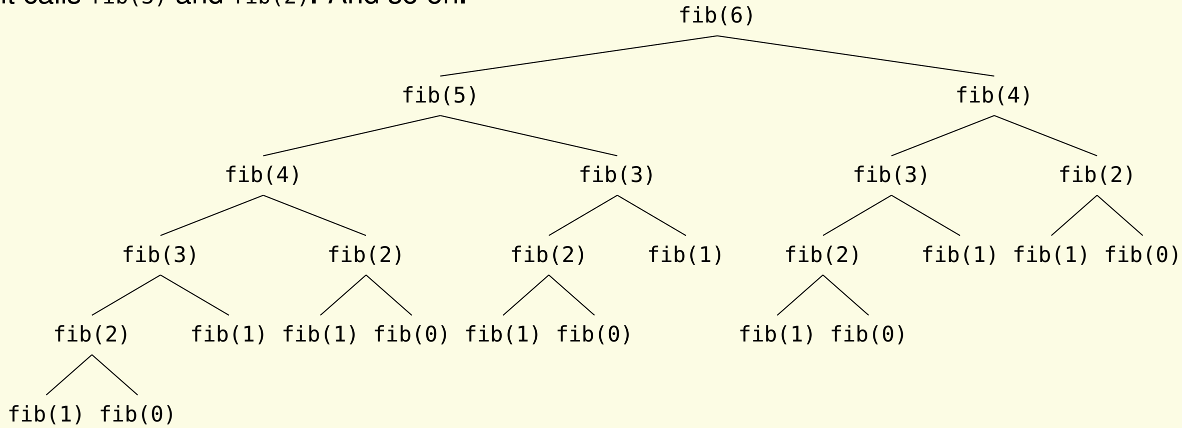
```
def fib(n: int) -> int:
    """Return the n-th Fibonacci number."""
    if n == 0: return 0
    elif n == 1: return 1
    else: return fib(n-1) + fib(n-2)
```

It works: `fib(6)` $\Rightarrow$ `8`, and `fib(10)` $\Rightarrow$ `55`. But what is `fib(38)`? It takes about 20 s to find.

`fib(55)` would take a day. `fib(67)` would take a year. `fib(82)` would take 1000 years.

This code is very slow. Why? Imagine we call `fib(6)`. It calls `fib(5)` and `fib(4)`.

But `fib(5)` also calls `fib(4)` and `fib(3)`. So we call `fib(4)` twice. And **each time** it is called, it calls `fib(3)` and `fib(2)`. And so on.



Example: the Fibonacci sequence

By hand, it's easy to calculate the sequence: just look at the previous two values:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

To calculate `fib(n)`, the answer is trivial if we already know `fib(n-1)` and `fib(n-2)`.

I'm going to rewrite my code so it stores the previous two values. Like so:

```python
def fib(n: int) -> int:
    """Return the n-th Fibonacci #."""
    f0 = 0
    f1 = 1
    while n > 0:
        new = f0 + f1
        f0 = f1
        f1 = new
        n = n - 1
    return f0
```

Example: the Fibonacci sequence

How can I do this using recursion?

```
def fib(n: int) -> int:
    """Return the n-th Fibonacci #."""
    f0 = 0
    f1 = 1
    while n > 0:
        new = f0 + f1
        f0 = f1
        f1 = new
        n = n - 1
    return f0
```

Problem: I need to give values to `f0` and `f1` before I start, and change them in each step.

Each recursive call only gets the values I pass as arguments.
So create parameters to store these values.

```
def fib_rec(n: int, f0: int, f1: int) -> int:
    """Return the Fibonacci # n steps
    after where it starts f0, f1.
    """
    if n > 0:
        return fib_rec(n - 1, f1, f0 + f1)
    else:
        return f0

fib_rec(100, 0, 1)
```

Implementing Loops with Recursion: a Procedure

Think about this transformation of code:

```
def fib(n: int) -> int:
    """Return the n-th Fibonacci #."""
    f0 = 0
    f1 = 1
    while n > 0:
        new = f0 + f1
        f0 = f1
        f1 = new
        n = n - 1
    return f0
```

```
def fib_rec(n: int, f0: int, f1: int) -> int:
    """Return the Fibonacci # n steps
    after where it starts f0, f1.
    """
    if n > 0:
        return fib_rec(n - 1, f1, f0 + f1)
    else:
        return f0
```

Procedure

To rewrite code using `while` so it uses recursion instead,

- Add a parameter for each local variable; set initial value in call: `fib_rec(n, 0, 1)`.
- Replace the `while` with a `if`.
- `return` the value from an recursive call with updated parameters.
- In the `else`, return the answer based on the parameter values.

Example: Square Roots

Procedure

- Add a parameter for each local variable; set initial value in call.
- Replace the `while` with a `if`.
- `return` the value from an recursive call with updated parameters.
- In the `else`, return the answer based on the parameter values.

Recall this code that estimates the square root:

```
def sqrt(n: float) -> float:
    """Approximate the square root of n.
    Requires: n >= 0"""
    g = 1.0 # initial guess; it may be bad, but it
    ## Don't start at the int 1; we promised to return a float.
    while abs(g**2 - n) > 0.0001:
        g = (g + n/g) / 2
    return g
```

Exercise

Use the procedure above to write `sqrt_rec` using recursion only.

Wrapper Functions

To use `fib_rec`, we need to remember to call it like `fib_rec(n, 0, 1)` instead of `fib(n)`.

Likewise, to use `sqrt_rec` we need to remember to call it like `sqrt_rec(n, 1.0)` instead of `sqrt(n)`.

For each, we could write a **wrapper function**:

```
def fib(n: int): return fib_rec(n, 0, 1)
def sqrt(n: float): return sqrt_rec(n, 1.0)
```

A **wrapper function** is a function that does some small setup before calling another function that does the main work. It may also do some cleanup afterwards.

This is OK, but it's kind of annoying.

I should write a docstring, annotations, and tests, even though they do almost nothing.

That's a lot of work for not much.

Default Parameter Values

We've seen before functions that don't always take the same number of arguments.

One example is `math.log`; it can be called with two arguments or one:

`math.log(81, 3)` $\Rightarrow$ `4.0`, corresponding to $\log_3 81$

`math.log(81)` $\Rightarrow$ `4.394449154672`, corresponding to $\log_e 81$

If we write `math.log(x)`, it is like writing `math.log(x, math.e)`.

This is what we want to do!

If we call `fib_rec(n)`, we want it to be like writing `fib_rec(n, 0, 1)`.

If we call `sqrt_rec(n)`, we want it to be like writing `sqrt_rec(n, 1.0)`.

Default Parameter Values

We only need one tiny change.

After the type annotation of a parameter, add `=val` to set a default value:

```
def fib_rec(n: float, f0: int=0, f1: int=1) -> int:
```

Now `fib_rec(n)` is like `fib_rec(n, 0, 1)`:

`fib_rec(100)` ⇒ `354224848179261915075`

Exercise

Make this one tiny change to your `sqrt_rec` function so you can call it without the extra `1`.

`sqrt_rec(100.0)` ⇒ `10.000000000139897`

Example: calculating cos

Recall this function to calculate the cosine that we wrote back in module 03:

```
def cos(x: float) -> float:
    """Approximate cos(x) using Taylor series."""
    total = 0.0
    i = 0
    sign = 1
    nextterm = sign * x**i / math.factorial(i)

    while abs(nextterm) >= 0.0001:
        total = total + nextterm
        i = i + 2      # count 0, 2, 4, 6, 8, ...
        sign = -sign   # alternate +, -, +, -, ...
        nextterm = sign * x**i / math.factorial(i)
    return total
```

Ex. Rewrite cos using recursion only.

Reminder:

Procedure

- Add a parameter for each local variable; set initial value in call.
- Replace the `while` with a `if`.
- `return` the value from an recursive call with updated parameters.
- In the `else`, return the answer based on the parameter

Mutable Default Parameter Values

There is one issue when the default parameter is mutable (e.g. a list or dictionary).

This looks good:

```
def reverse_rec(L: list[any], answer: list[any]=[]) -> list[any]:
    """Return a list containg all values from L, in reverse order.  Mutates L to []."""
    if L != []:
        newval = L.pop()
        answer.append(newval)
        return reverse_rec(L, answer)
    else:
        return answer
```

`reverse_rec([2,4,6,0,1])` ⇒ `[1, 0, 6, 4, 2]`

Looks good so far. But call the function a second time:
`reverse_rec([7])` ⇒ `[1, 0, 6, 4, 2, 7]`

This second call does not start with a new `answer=[]`, but the **same list**.

One Weird Trick

This code fixes it:

```
def reverse_rec(L: list[any], answer: list[any] | None=None) -> list[any]:
    """Return a list containg all values from L, in reverse order.  Mutates L to []."""
    if answer == None: # The "default value":
        answer = []

    if L != []:
        newval = L.pop()
        answer.append(newval)
        return reverse_rec(L, answer)
    else:
        return answer
```

Procedure

We change two things:

1. The default value for the parameter is `None`. (And add "`| None`" to the annotation.)
2. When `answer` is `None`, we assign a new list to it.

One Weird Trick

```
def fib_sequence(n: int,
                 seq: list[int]=[0,1]) -> list[int]:
    """Return the first n terms of the Fibonacci sequence."""
    if n > 2:
        seq.append(seq[-2] + seq[-1])
        return fib_sequence(n-1, seq)
    else:
        return seq
```

`fib_sequence(6)` ⇒ `[0, 1, 1, 2, 3, 5]`

Looks good, but call it again:
`fib_sequence(6)` ⇒ `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]`

Ex. Fix `fib_sequence`.

Procedure

We change two things:

1. The default value for the parameter is `None`. (And add "`| None`" to the annotation.)
2. When `answer` is `None`, we assign a new list to it.

Python hates recursion

Any computation that can be done can be done using only recursion.

However, Python is not designed to run recursive code well.

Try running `countdown(5000)`.

By default, it can recurse to a depth of only 3000 or so.

For most reasonable situations, this is not a serious limitation. A `Family` of depth 3000, where everyone had 2 children, would have $2^{3000} \approx 10^{900} = \text{googol}^9$ people. That's vastly larger than any dataset that can exist in the universe.

If a computation is very "deep", it's probably better to write using a loop.

Recursion is great with tree-like data. In Python, we should use loops for most other purposes.

We need to be able to use recursion. We can't claim to be skilful programmers without it.

These exercises review a pile of skills from throughout the term.

Exercise

Write a function `times_table_iterative(n:int) -> list[tuple[int,int,int]]` that creates the times table up to `n×n`. **Do not use recursion.** Use two `for` loops.

```
check.expect("3x3", times_table_iterative(3),
    [(1,1,1),(1,2,2),(1,3,3),(2,1,2),(2,2,4),(2,3,6),(3,1,3),(3,2,6),(3,3,9)])
```

Exercise

Rewrite `times_table_iterative` so it uses two `while` loops.

Exercise

Consider how you can re-write the inner `while` loop as a separate recursive function `times_table_row(n, ans, i, j=1)`.

Write a function `times_table_recursive` so it uses no loops. Call `times_table_row` repeatedly, using recursion.

- Be able to write recursive code to draw fractals.
- Work with recursive data structures, especially trees and including leaf-labelled trees.
- See how we can use recursion instead of loops to do computations.

Before we begin the next module:

- Read and complete the exercises in module 9 of the online textbook, at https://online.cs.uwaterloo.ca/
- Complete the module 9 Review Quiz, due on Monday.