# Day 3

## Control-Flow, Functions, Good code

# Recap

- Booleans
- If Statements
- Lists
- Dictionaries
- For and While Loops
- List comprehensions
- Be careful with mutables and assignments!

# Today’s menu

- Practice exercises from yesterday
- Functions and Scope
- Concepts of control-flow & Good code: Or at least not bad.
- Documentation: No, cursing in the comments doesn’t count.
- Problem solving 101: Now with 10% more strategy!

# But first: Exercises to recap

- Write a script that takes a number and prints the sum of the squares of all integers less than that number.
  (Example for 5: 4**2 + 3**2 + 2**2 + 1**2 = 30)
  Bonus points for doing it in one line.
- Write a script that checks if there are lowercase characters in a string. Then print only the lowercase characters in the string.

# Functions

Can be seen as mini-programs

Used to chop program into logical units

Offer defined inputs and outputs

Functions can call other functions

Functions can call themselves (recursion)

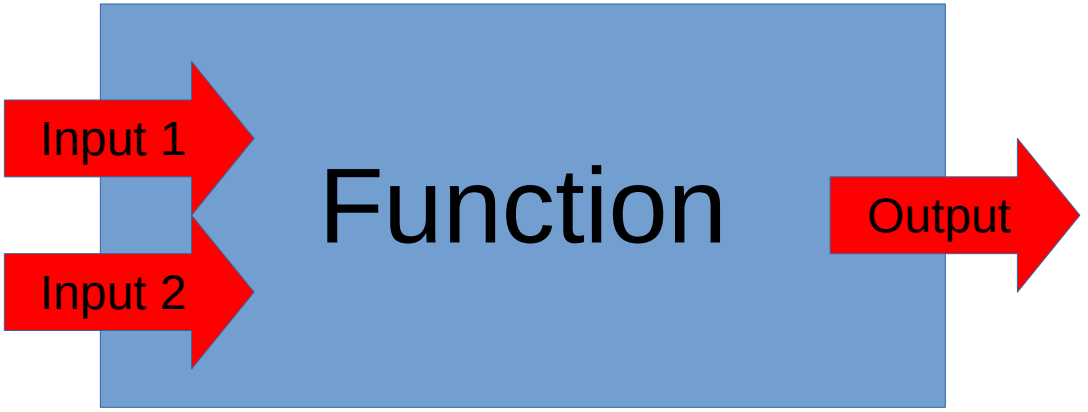
```
def function(input1, input2):
    do_stuff
    return output
```

# Exercises

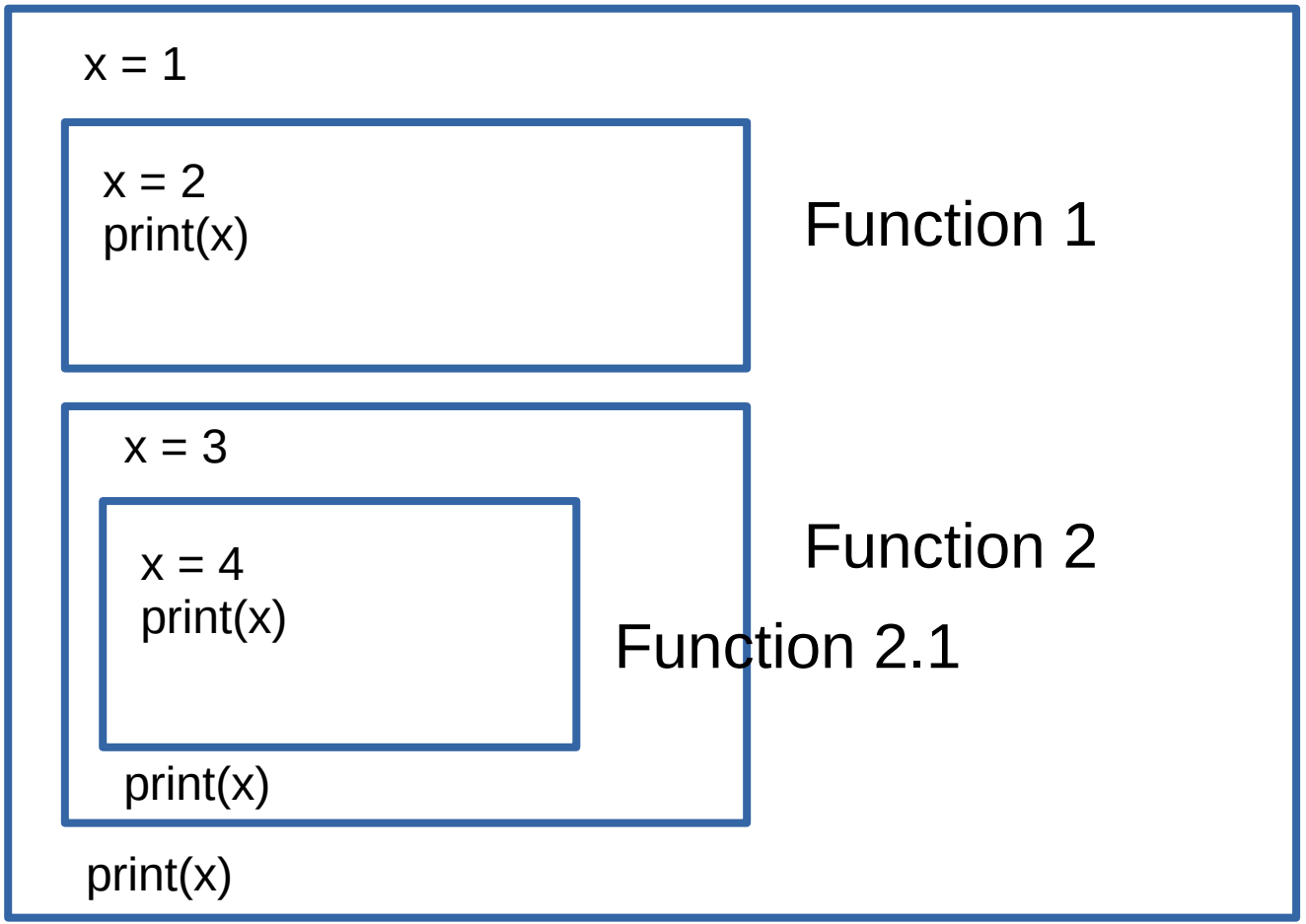
- Write a function that calculates the mean of a list of numbers.
- Write a function that returns a string input in inversed case.
- Write a function that calculates the factorial of any number.
  (The factorial of n equals (1 * 2 * … * n-1 * n)

# Scope

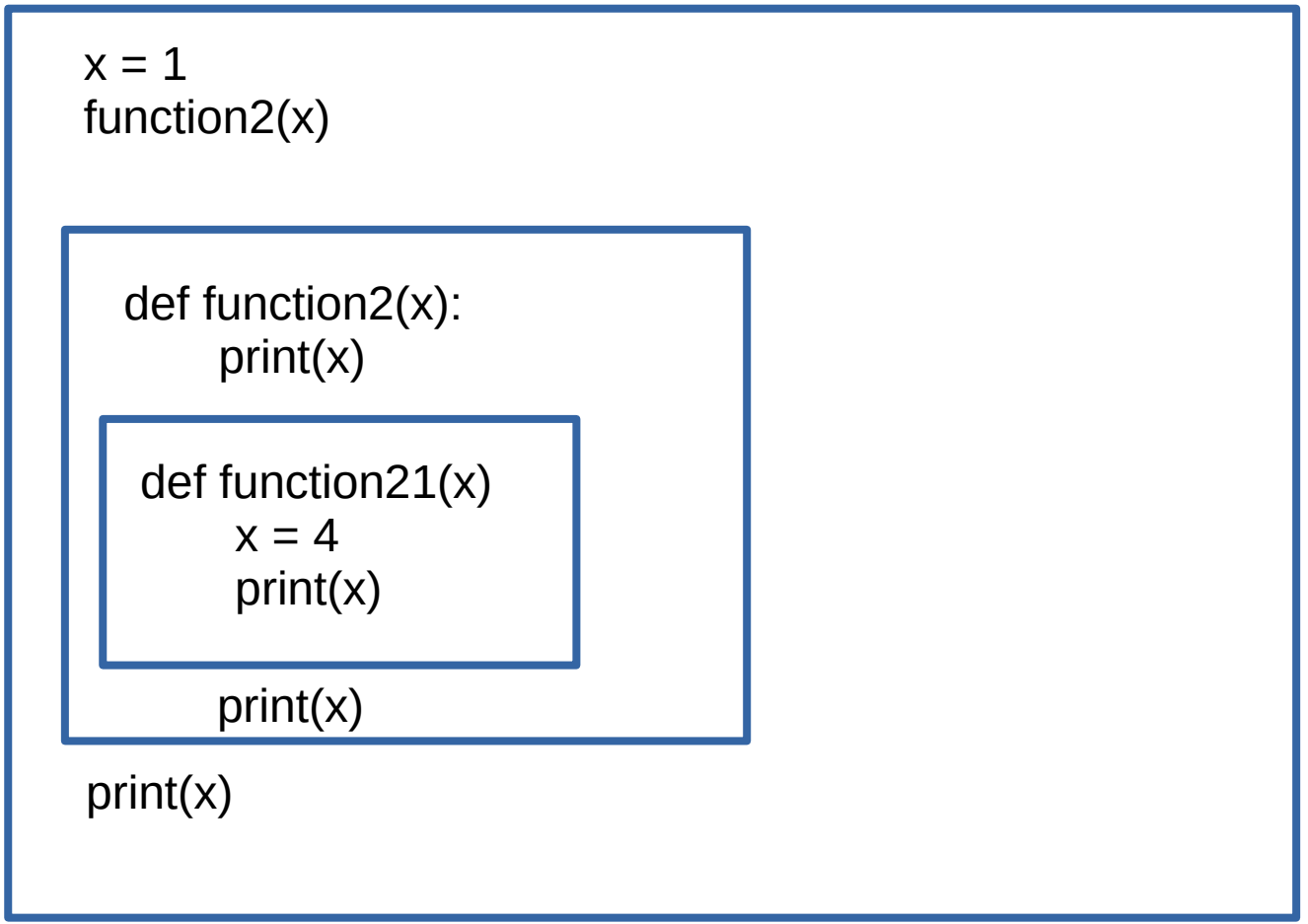
Here be dragons



# Scope

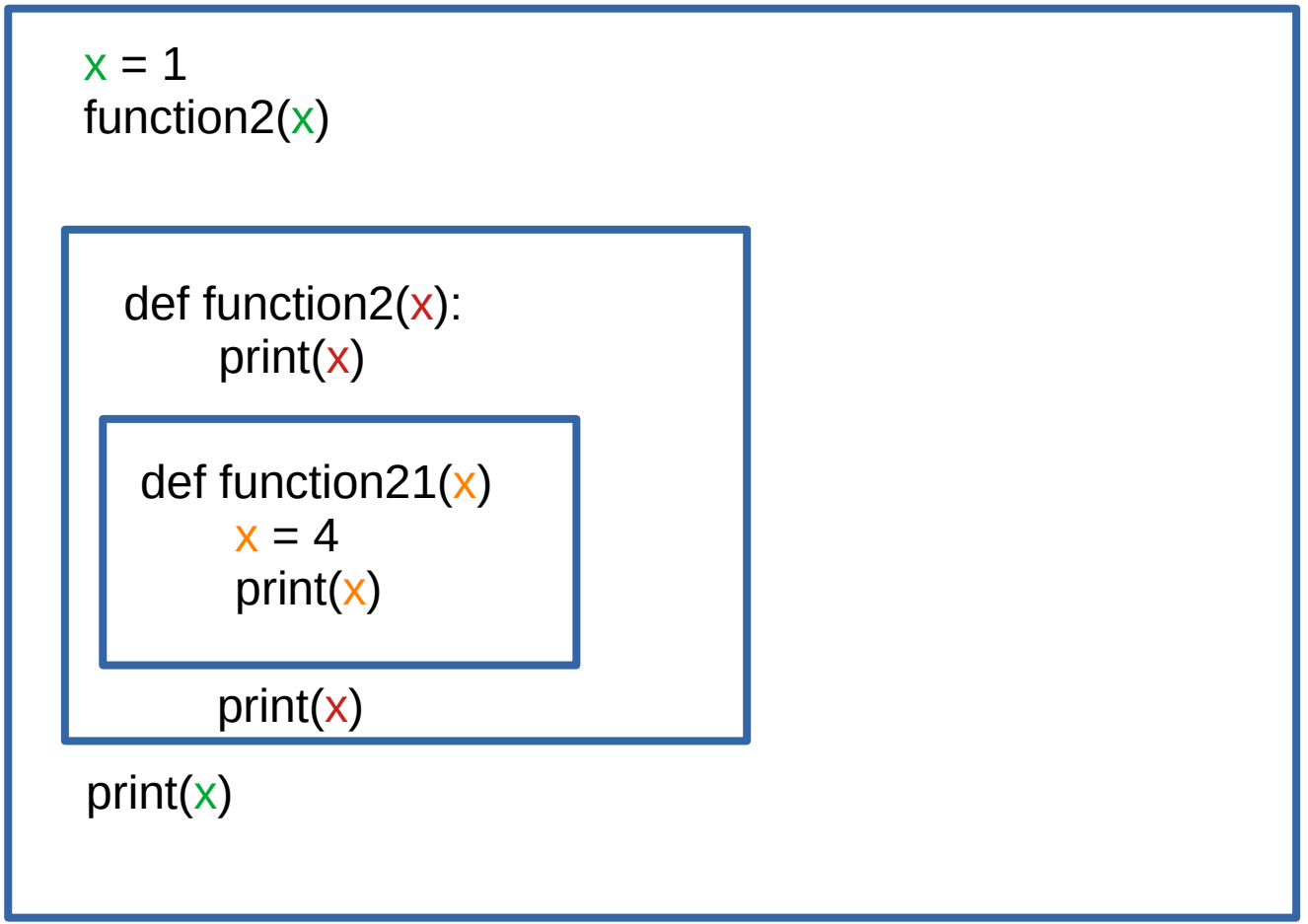

x = 1
x = 2
print(x)
Function 1
x = 3
x = 4
print(x)
Function 2.1
print(x)
print(x)
Main program

# Scope



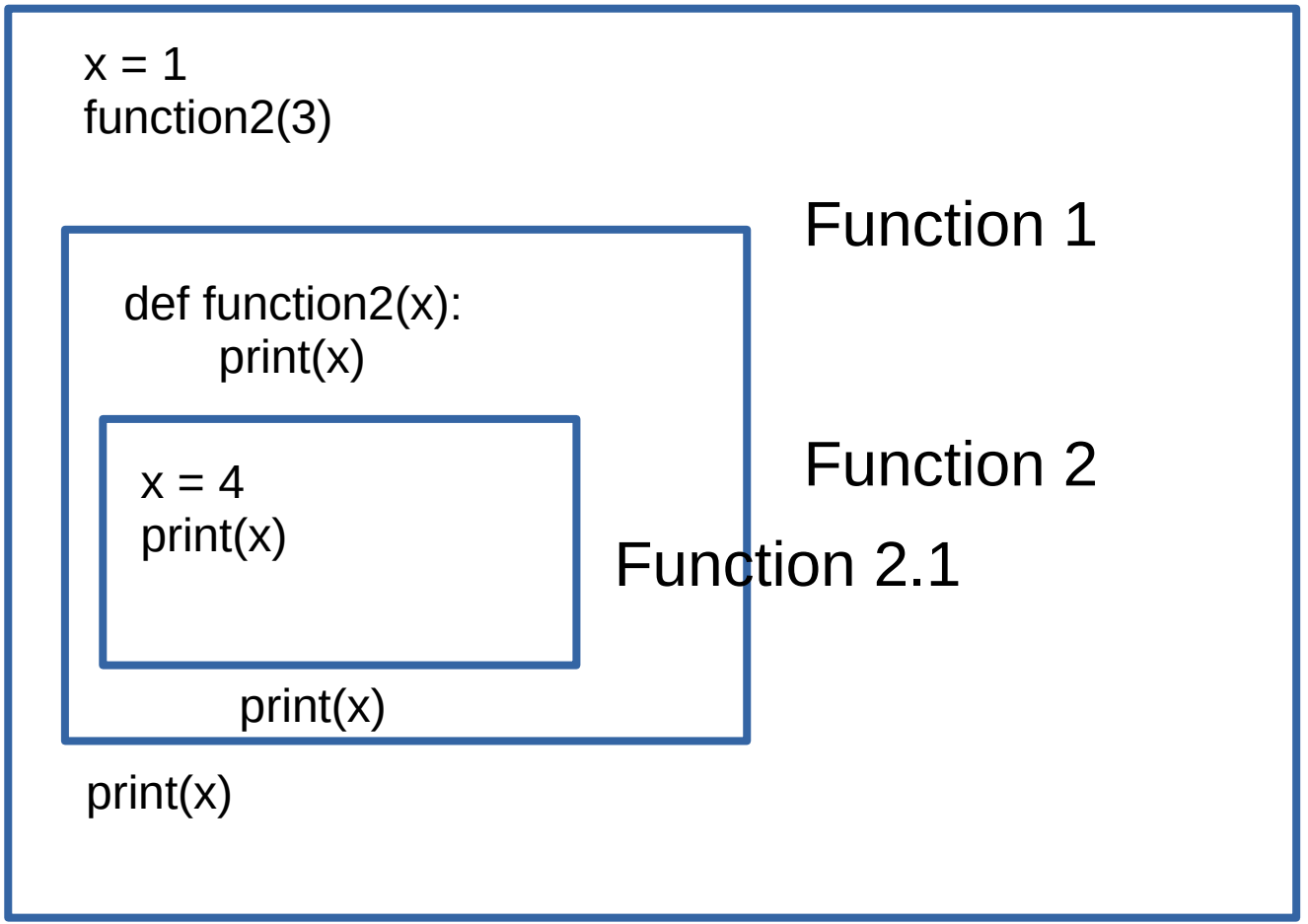

Main program

# Scope




Main program

# Scope


x = 1
function2(3)
Main program
Function 1
def function2(x):
print(x)
Function 2
x = 4
print(x)
Function 2.1
print(x)
print(x)

# Scope - Summary

Every function is independent

Do NOT use confusing naming schemes

Pick smart variable names when using functions

# Exercise

- Create 3 functions calculating the ___ of a list of numbers:
  - arithmetic mean
  - geometric mean
  - harmonic mean
- Now write one function that unites them into one and chooses which one to pick using an input parameter. Set a default using: def function(param1, param2=’default’)

# Advantages of using functions

Makes code easy to read and understand

Clearly defines components

Functions are isolated, finished sections of code

DRY – Don’t Repeat Yourself

Clearly defined in- and output

# Refactoring

**Refactoring:**
Spinning code off into functions and restructuring

Always becomes necessary in larger projects

Don’t push it off until the end

Keep the original and the refactor for checks

# Good code

Good code is ...

- easily read and understood
- structured and logical
- concise
- easily maintained
- well documented

# Points to look out for

```
def average(list):
    return (sum(list) / len(list))
```

```
a1 = sum([l**2 for l in l1]) / len(l1)
a2 = sum([l**2 for l in l2]) / len(l2)
```

```
list_1 = [element**2 for element in list_1]
list_2 = [element**2 for element in list_2]
```

```
avg1 = average(list_1)
avg2 = average(list_2)
```

# What makes code structured?

DRY: Don’t repeat yourself

Instead of repeating code, make it a function

Refactor!

Many short functions are better than few large

Visibly structure code

Write blocks doing similar things

# Disclaimer: “Real” code

Often it’s difficult to know if code will be reused

Quick and dirty code often turns important

Be ready to rewrite stuff when it becomes necesary

Every scientific coder struggles with this

# What makes maintenance easy?

**Scenario:**
You return to your code in n Years / at the end of your thesis / at the end of your PhD. You know you have to change one aspect. How long does it take?

# What makes maintenance easy?

Hardcoding:

Code that works for one specific scenario only

Limit it to a minimum

If you have to: Collect it all, prominently, in one point

# Documentation

Explains what your code does

Helps other people **and you**

- Write a Readme file
- Explain script shortly at the start
- Explain every function
- Comment difficult sections

# Docstring

“””

Text in triple quotes at start of Function or script

Explains what the function does

Names and explains all inputs

Names and explains all outputs

“””

# Docstring

```
def average(number_list):
    """
    Returns the arithmetic average of a list of numbers.
    :param number_list: A list of integers or floats.
    :return: Average of the list of numbers. Integer or float.
    """
    total = sum(number_list)
    length = len(number_list)
    return (total / length)
```

# Mini-project

- Professional and extended FizzBuzz

  You've created *FizzBuzz* (FB). But what if you want *FizzBuzzCrackle*, or ShooZoom*WhackPshht* ?
  Write an FB clone that takes Numbers as Inputs and behaves like Fizzbuzz, replacing numbers by the sounds they're divisible by. For example, an Input of 2: "Whack", 3: "Zip" 4: "Yow" would print 24 as "WhackZipYow". The number range can be picked freely.

# How to solve a coding problem:

- Really understand the problem!
- Take it apart into small (!) components
- For each component, ask yourself:
  - What structures would work for that component?
  - What kind of decisions do I have to make?
    What kind of path does my code have to take?

# Step 1

- Explain the problem to your seat neighbor in simple terms.
- What kind of Inputs do I get?
- What kind of Outputs do I want?

# Discussion

- Input: Most sensibly a dictionary of {Integer: “String”}, also possibly a list of tuples or something similar. Also a range of numbers.
- Output: Print strings to the console

# Step 2

- What tasks can my problem be divided into?
- Make your own division into subtasks first.
- Then discuss with your neighbor and the rest.
- When you have your results, write them down as *Pseudocode*.

# Pseudocode

Simple instructions of what your code needs to do

In plain everyday language

Has a *coding sound* to it

Give students coding task

For each student:

- Look at their code
- Tell them they’re doing well
- Make constructive suggestions

# Subtasks

- Bring the Input into a form we can use it.
- Go over the number range, for each number
  - Check divisibilities
  - Make string
  - Print string

# Subtasks

- Bring the Input into a form we can use it.
- Go over the number range, for each number in range (NIR)
  - For each sound-number pair:
    - Check if NIR is divisible by number in pair (NIP)
    - If NIR is divisible by NIP: add sound to string
  - Make string
  - Print string

# Subtasks

- Bring the Input into a form we can use it.
- Go over the number range, for each number in range (NIR)
  - Make an empty string
  - For each sound-number pair:
    - If NIR is divisible by number in pair (NIP)
      - add sound to string
  - If the string is empty:
    - Print the number
  - Else:
    - Print the string

# Go forth and code!

- And don’t forget to document and comment.
- Think about which aspects may be functions and spin them out.
- But don’t worry too much. For now, just code and get things working!

# Make it high quality!

- Find problematic spots in your code
- Make it *good code*
- After 5 minutes: Present your code to your neighbor.
  Explain your reasons for doing things the way you did them.

# Errors - not just an annoyance

Errors point out where you’re wrong

Sometimes you can expect errors:

Wrong input
Missing data
...

Handling Errors can be crucial

# Try - Except

Attempts to execute code under try

If *try* code crashes, executes *except* code

Needs specification of Error type

```
def average(number_list):
    total = 0
    for number in number_list:
        try:
            total += number
        except TypeError:
            print(‘Element is not a number’)
```

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Not a valid number.  Try again...")
```

# Error types

- SyntaxError
- IndexError
- KeyError
- TypeError
- ValueError
- NameError
- ZeroDivisionError
- ...

# Exercises - Errors

- Write a function that transcribes DNA into RNA.
- Think about what problems could occur when someone uses the function.
- Write some problematic inputs for the function. Basically try to wreck your own code as much as possible.
- Try to account for your problematic inputs.
- Discuss problems and solutions with your neighbor.
- Have your neighbor design problematic inputs for your code.

# Throwing Errors

It can be useful to purposely throw errors

Crashes make mistakes obvious

Don’t let code run to produce wrong results