4

# Lecture 4

Abstraction with Functions and Imports

# Before the new stuff!

A quick recap of functions, and some more about recursion!

# Function Recap

Function definition

Function name

Function argument(s)

```
def name_and_greet(my_name):
    print("Hello! What is your name?")
    your_name = input()
    print("Hello,", your_name, "! My name is", my_name)
    return your_name

student_list = []
for i in range(5):
    student_name = name_and_greet("Lloyd")
    student_list.append(student_name)
```

Function body

Function return

Rest of program (the “main” function)

Function call

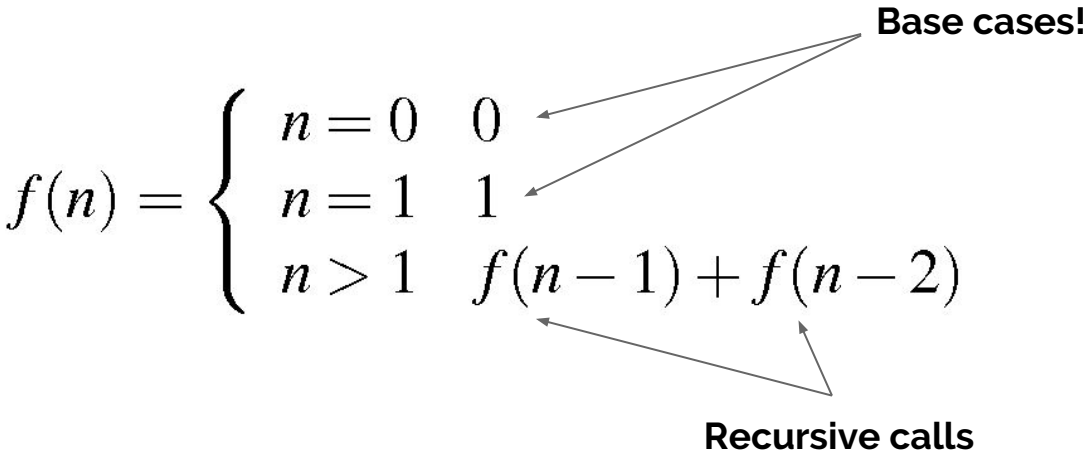
# Recursion Recap

*Calling a function from within its own definition!*

# Recursion: Fibonacci

$$f(n) = \begin{cases} n = 0 & 0 \\ n = 1 & 1 \\ n > 1 & f(n-1) + f(n-2) \end{cases}$$

# Recursion: Fibonacci



# Recursion: Fibonacci

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

# Recursion: Fibonacci

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

$$f(n) = \begin{cases} n = 0 & 0 \\ n = 1 & 1 \\ n > 1 & f(n-1) + f(n-2) \end{cases}$$

# Recursion: Fibonacci

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Base cases!

$$f(n) = \begin{cases} n = 0 & 0 \\ n = 1 & 1 \\ n > 1 & f(n-1) + f(n-2) \end{cases}$$

# Recursion: Fibonacci

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Base cases!

Recursive calls

$$f(n) = \begin{cases} n = 0 & 0 \\ n = 1 & 1 \\ n > 1 & f(n-1) + f(n-2) \end{cases}$$

# Recursion

*As with loops, we have to be careful that we terminate*

# Recursion

***As with loops, we have to be careful that we terminate***

*This is almost always done by being careful with our* ***base cases***

# Recursion vs Loops

## Recursion

- Nicer to write
- Closer to the mathematical definition of some functions
- Need to be careful with base case(s)
- Often times, slower, less memory/space efficient (not always! Look into “tail recursion” if you’re interested!

## Loops

- Can be difficult to write
- Closer to how your computer actually does things
- Need to be careful with loop conditions
- If written well and optimized, often faster and more space-efficient

# Recursion/Loop Example: Reversing a List

# What is abstraction?

A way of ***reducing complexity*** within a program, to make it easier to reason about.

# Why do we do it?

- Readability
- Reusability
- Extensibility

# Where have we seen it?

Functions! Functions are one of the primary ways we *abstract* logic away from sections of our code.

# What are you talking about?

So far this concept seems very *abstract*. Let's see it in practice (just using functions).

# Abstraction for readability/reusability

```
my_list = [1,2,3,4]
print(my_list)
print("How many new elements?")
number_of_elements = int(input())
for i in range(number_of_elements):
    new_element = int(input())
    my_list.append(new_element)
length = len(my_list)
for i in range(length // 2):
    temp_value = my_list[i]
    my_list[i] = my_list[-(1 + i)]
    my_list[-(1 + i)] = temp_value
for i in range(5):
    new_element = int(input())
    my_list.append(new_element)
```

```
for i in range(length // 2):
    temp_value = my_list[i]
    my_list[i] = my_list[-(1 + i)]
    my_list[-(1 + i)] = temp_value
new_list = my_list[2:]
length = len(new_list)
for i in range(length // 2):
    temp_value = new_list[i]
    new_list[i] = new_list[-(1 + i)]
    new_list[-(1 + i)] = temp_value
number_of_elements = int(input())
for i in range(number_of_elements):
    new_element = int(input())
    new_list.append(new_element)
print(new_list)
```

# Abstraction for readability/reusability

```
my_list = [1,2,3,4]
print(my_list)
print("How many new elements?")
number_of_elements = int(input())
for i in range(number_of_elements):
    new_element = int(input())
    my_list.append(new_element)
length = len(my_list)
for i in range(length // 2):
    temp_value = my_list[i]
    my_list[i] = my_list[-(1 + i)]
    my_list[-(1 + i)] = temp_value
for i in range(5):
    new_element = int(input())
    my_list.append(new_element)
```

```
for i in range(length // 2):
    temp_value = my_list[i]
    my_list[i] = my_list[-(1 + i)]
    my_list[-(1 + i)] = temp_value
new_list = my_list[2:]
length = len(new_list)
for i in range(length // 2):
    temp_value = new_list[i]
    new_list[i] = new_list[-(1 + i)]
    new_list[-(1 + i)] = temp_value
number_of_elements = int(input())
for i in range(number_of_elements):
    new_element = int(input())
    new_list.append(new_element)
print(new_list)
```

```
my_list = [1,2,3,4]
print(my_list)
append_new_elements(my_list)
reverse(my_list)
append_n_elements(my_list, 5)
reverse(my_list)
new_list = my_list[2:]
reverse(new_list)
append_new_elements(new_list)
print(new_list)
```

# Your turn

- Make a program (with functions and abstraction!) that will:
  - Ask for a number, n
  - Ask for n more numbers and put them in a list
  - Reverse this list
  - Add 5 to every number in this list
  - Ask for a number, m
  - Ask for m more numbers and put them in a list
  - Add 26 to every number in this list
  - Reverse this list
  - Print both lists

# Imports and Libraries

What if someone else has already made the function we want?

Even if they haven't, we might not want to fill up our program with function definitions…

# Import Statements

```
import abstraction_functions


my_list = [1, 2, 3]
abstraction_functions.reverse(my_list)
```

# Import Statements

```
import abstraction_functions


my_list = [1, 2, 3]

abstraction_functions.reverse(my_list)
```

*Module* name

Import keyword

# Import Statements

```
import abstraction_functions


my_list = [1, 2, 3]

abstraction_functions.reverse(my_list)
```

*Module* name

Import keyword

*Module* name

# Import Statements

```
import abstraction_functions

my_list = [1, 2, 3]

abstraction_functions.reverse(my_list)
```

*Module* name

Import keyword

*Module* name

*Function* name

# Import Statements

```
from abstraction_functions import reverse


my_list = [1, 2, 3]
reverse(my_list)
```

# Import Statements

*Module* name

```
from abstraction_functions import reverse

my_list = [1, 2, 3]
reverse(my_list)
```

from … import keywords

# Import Statements

*Module* name

```
from abstraction_functions import reverse

my_list = [1, 2, 3]
reverse(my_list)
```

*Function* name

from … import keywords

# Import Statements

*Module* name

```
from abstraction_functions import reverse

my_list = [1, 2, 3]
reverse(my_list)
```

*Function* name

from … import keywords

*Function* name

# Import Statements

*Module* name

```
from abstraction_functions import reverse

my_list = [1, 2, 3]
reverse(my_list)
```

*Function* name

from … import keywords

*Function* name

Note that we no longer do "`abstraction_functions.reverse()`"