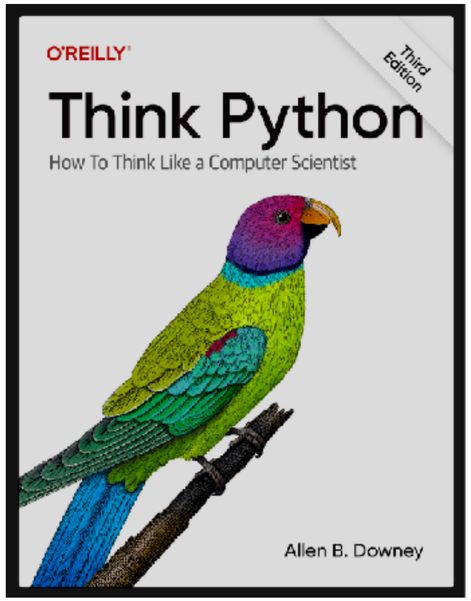
COL1000: Introduction to Programming

Functions

Subodh Sharma | Lec 21 | Oct 07

Announcement

- Additional reference book:
  - https://allendowney.github.io/ThinkPython/
- **Next Week Lab Test on Functions - be prepared!**



Functions: Lambda (RECAP)

- **Lambda functions**: They are anonymous functions (i.e. don't have a user-specified name)
  - Syntax: `lambda arguments: expression`
    - Arguments can be 0 or more, but **have a single expression** whose **results are returned**!
    - E.g., `add_five = lambda x: x+5; print(add_five(7)) #12`
    - E.g., `lst = list(map(lambda x: x**2, [1,2,3])) # [1,4,6]`

Functions: Lambdas — where can I use them?

Use in a map

```
squared = list(map(lambda x: x ** 2, numbers))
```

Used to implement filter

```
Data = [2, 5, 15, 17, 4, 29]

Filtered = list(filter(lambda x: x > 10, Data))
```

In general, wherever simple one-lined function suffices!

Functions: Closures (RECAP)

- A **closure** is a function object that **remembers values from its enclosing scope even after that scope has finished its execution**

```
def make_mean_for_streaming_inputs():
    total = 0.0
    n = 0
    def add(x):
        nonlocal total, n
        total += x; n += 1
        return total / n
    return add

avg = make_mean_for_streaming_inputs()

print(avg(10)); print(avg(14)); print(avg(16))
```

- **`return add`:** closure cell is formed
  - **`total`** and **`n`** are *captured*
- Each call to `avg` remembers the values of **`total`** and **`n`**

bject

Functions: Typing Hints (RECAP)

- Provide explicit typing annotations to help with
  - readability,
  - Static analysers like mypy etc.

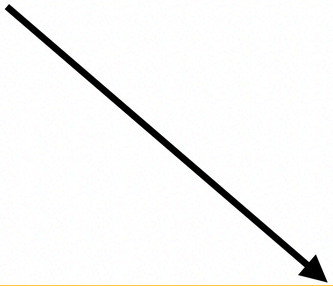
```
def add (x:int, y:int) --> int:
    return x + y
```

After arrow, the type of returned value

Functions: Typing Hints (RECAP)

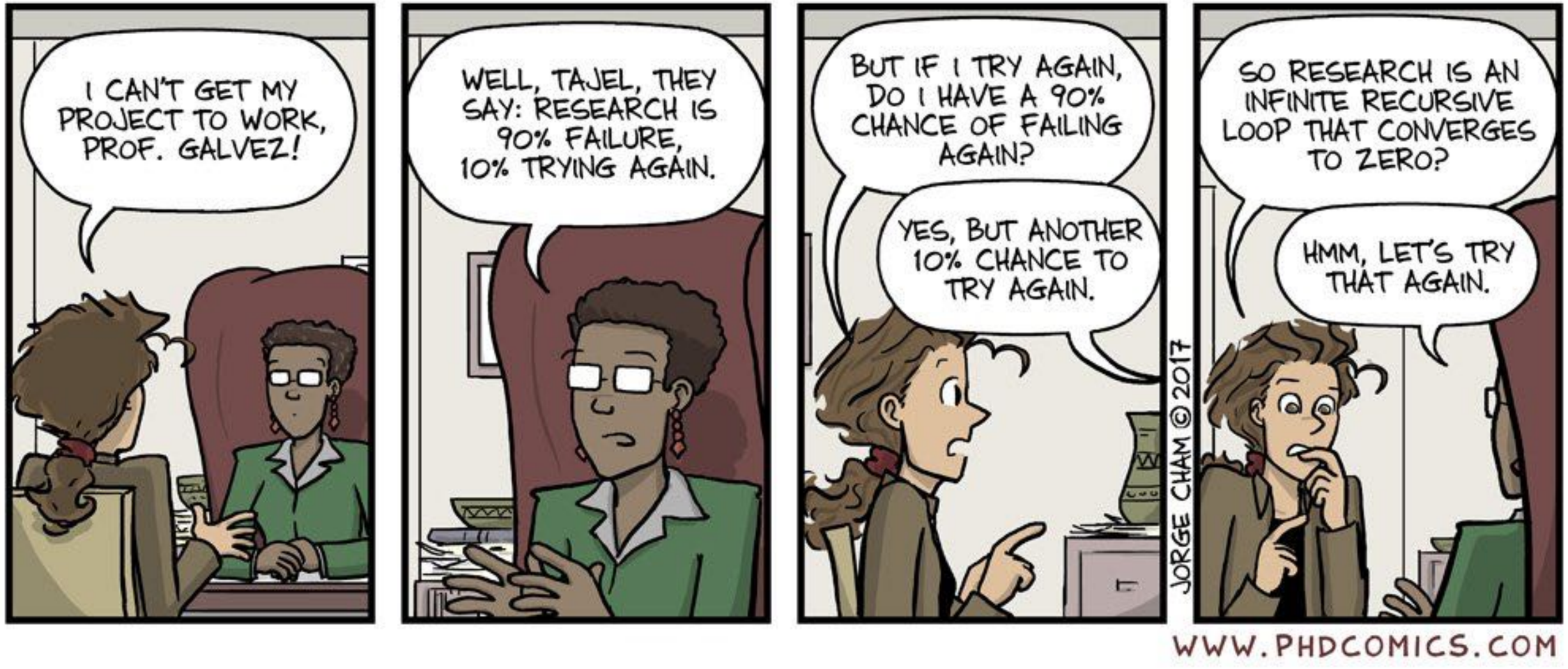
- Provide explicit typing annotations to help with
  - readability,
  - Static analysers like mypy etc.

```
from typing import Callable
def make_adder(k:int) -> Callable[[int], int]:
    ...
    def add(x:int = 2)->int:
        ...
    return add
```



Callable[[arg1, arg2, ..], [return]]

Recursion


I CAN'T GET MY PROJECT TO WORK, PROF. GALVEZ!
WELL, TAJEL, THEY SAY: RESEARCH IS 90% FAILURE, 10% TRYING AGAIN.
BUT IF I TRY AGAIN, DO I HAVE A 90% CHANCE OF FAILING AGAIN?
YES, BUT ANOTHER 10% CHANCE TO TRY AGAIN.
SO RESEARCH IS AN INFINITE RECURSIVE LOOP THAT CONVERGES TO ZERO?
HMM, LET'S TRY THAT AGAIN.
JORGE CHAM © 2017
WWW.PHDCOMICS.COM

What is Recursive Style of Programming

- A paradigm where the function, in its body, calls itself repeatedly
- Classic example:

```
def factorial(n:int) -> int:
    if n == 0: return 1
    elif n < 0:
        raise Error("Not defined on neg values")
    else: return n * factorial(n-1)
```

What is Recursive Style of Programming

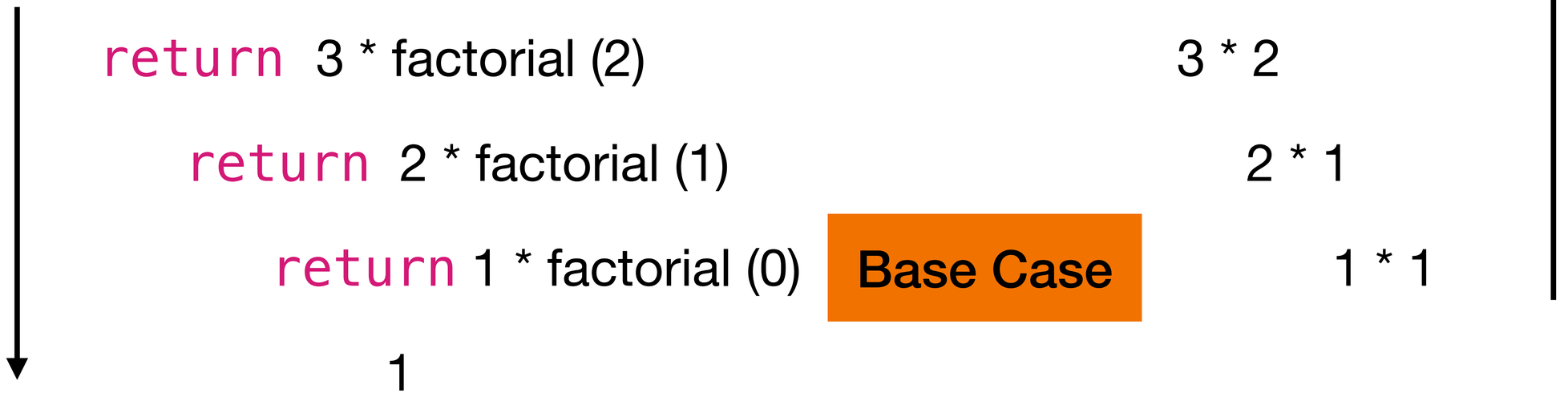
Mental Model

```
problem(n) -> func (problem(n-1)) -> … -> Base Case
```

Key Takeaway: Each recursive call takes the problem closer to the base case!

Example Recursion: Computation View

- factorial(4): `return` 4 * factorial (3)



Example Recursion: Stack Memory View

- **push** factorial (4)
- **push** factorial (3)
- **push** factorial (2)
- **push** factorial (1)
- **push** factorial (0) —> returns 1 (**pop**)
- factorial(1) —> returns 1*1 = 1 (**pop**)
- factorial(2) —> returns 2*1 = 2 (**pop**)
- factorial(3) —> returns 3*2 = 6 (**pop**)
- factorial(4) —> returns 4*6 = 24 (**pop**)

More Examples of Linear Recursion

- Sum the list recursively
  - Show the limits of stack!
- Converting a non-negative integer to binary
  - How can you optimise?
- How do you establish the correctness of your recursive formulation?
  - **Fun fact:** In recursive formulation, correctness is embedded in program development!

Sum the list recursively

```
from typing import Sequence

def recSum (lst: Sequence[int]) -> int:
    #specify the base case
    if lst == []:
        return 0
    #recursive case
    else lst[0] + recSum(lst[1:])
```

For Typing Hints

func(problem (n-1))

Correctness Proof?

Co-Develop: Binary Search

- Given a **list of sorted** elements and an element **x**
  - Find whether x exists in the list and if so where?
- Algorithm:

  1. Search x at the mid of the list: If found then return the index, else go to step (2)
  2. If the element at mid > x, then lst = lst[mid+1:] and go to step 1
  3. Else, lst = lst[:mid] and go to step 1

Functions: Exceptions

- Exceptions in functions **denote errors or unexpected conditions**
- There are two ways of handling exceptions in the code:
  - Explicitly **raising the errors with an appropriated exception message**
  - **Catching** them to handle them gracefully
  - Let us see examples of each!