CS 131 - Chapter 3

Part 1: Funtions Cont.

Review

- Calling Functions
- Defining Funcions

Today

- Functions
    - Abstraction
    - Decomposition
- Namespace

Functions - review:

- simply: a package/block of code (a set of statements)
- that can be called repeatedly with different inputs (arguments/parameters)
- to give output
- or do some action each time it is called

Why Functions

- Code Reuse:
    - Functions allows us to group and generalize code ot be used arbitrarily many times after it is defined
- Procedural Decomposition:
    - Functions also provide a tool for splitting systems into pieces that have a well-define role: One functios for each subtask.
- Easier Code to Maintain & Update:
    - Having code implemented once and calling it allows easy code maintainance and debugging, and also adding new feartures

Exercise: What is one main purpose of a function:

1. To improve the speed of execution
2. To help the programmer organize programs into chunks that match how they think about the solution to the problem.
3. All Python programs must be written using functions
4. To calculate values.

Abstraction:

separation of concerns:

- Do not need to know how something works from inside
- Use devices as black-boxes
- feed the black-box with **valid input** - and - get out a **expected output**

Example:

Projector:

- plugin the right cable with the right connection to feed signal from the right device
- get the output on the screen

Decomposition:

different pieces work together to provide the final solution

Example:

- using `sum()` `sqrt()` `add()` `cos()` `sin()` together to do some calculations

`def` Statement

```
def <name> (arg1,arg2, ... argN):
   <statements>
   return <value>
```

- The `def` statement creates a function object and assigns it to a name.
- Python interpreter executes the function bodyeach time the function is called.
- The argument names will be assigned with the objects passed in the parenthesis at the point of a call.
- return ends function call and sends a result back to a caller

```
Priniting from inside is_even()
False
```

Scope Rules

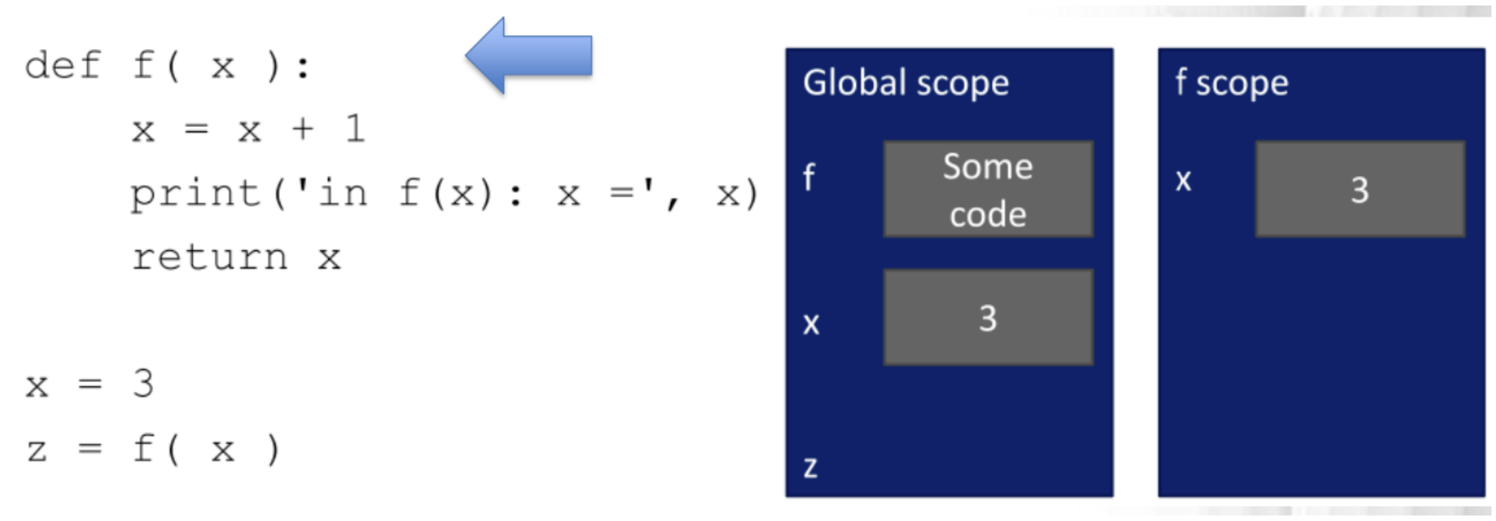
- **Namespace** is the place where names live
- The location of name's assignment defines the **scope** of the name visibility.
- Names defined inside a `def` can be seen only by the code inside the def.
- Names defined inside a `def` do not clash with variables outside the `def`, even if the same name is used elsewhere.

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [49], in <cell line: 1>()
----> 1 print(t)

NameError: name 't' is not defined
```

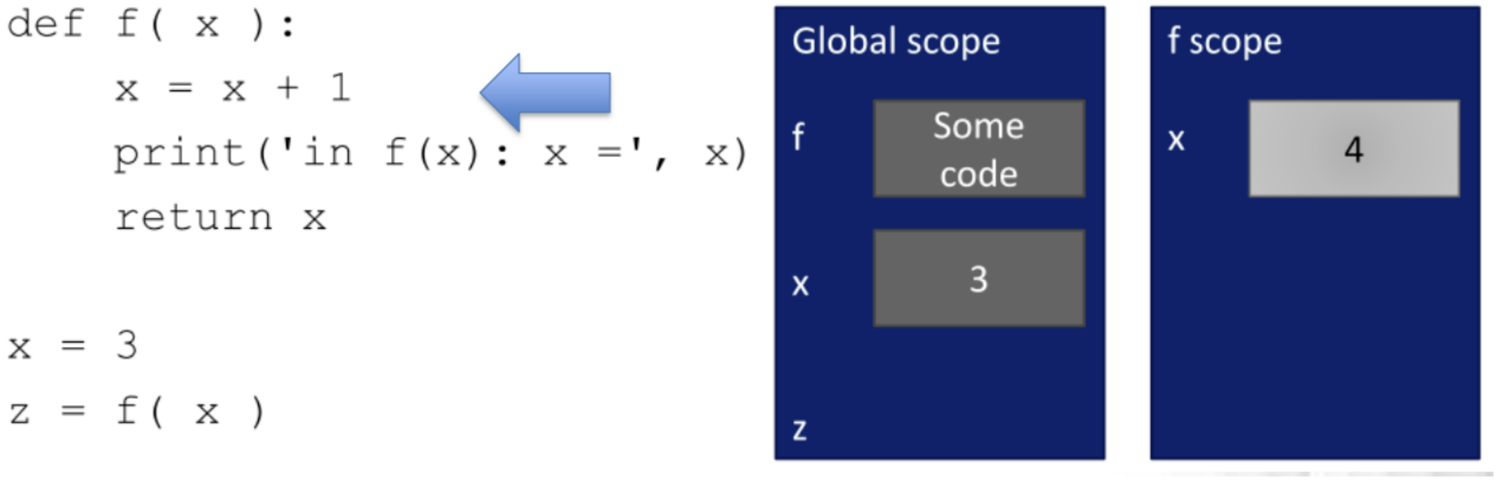
Example

```
inside fun1(x): x = 4
outside fun1(x): x = 3
```



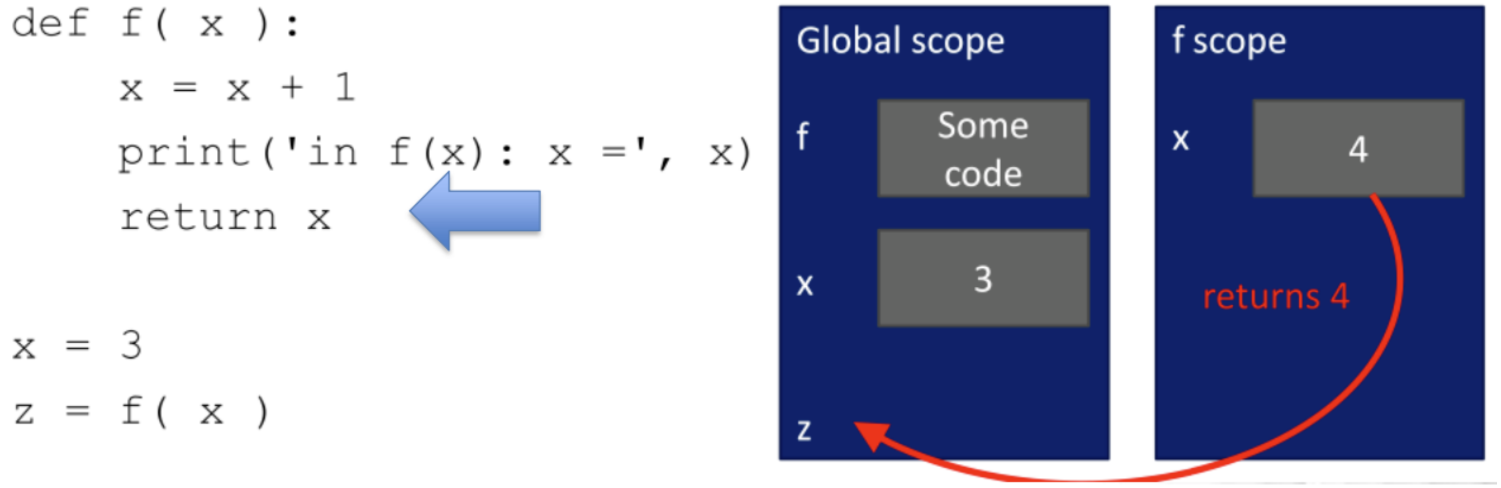
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



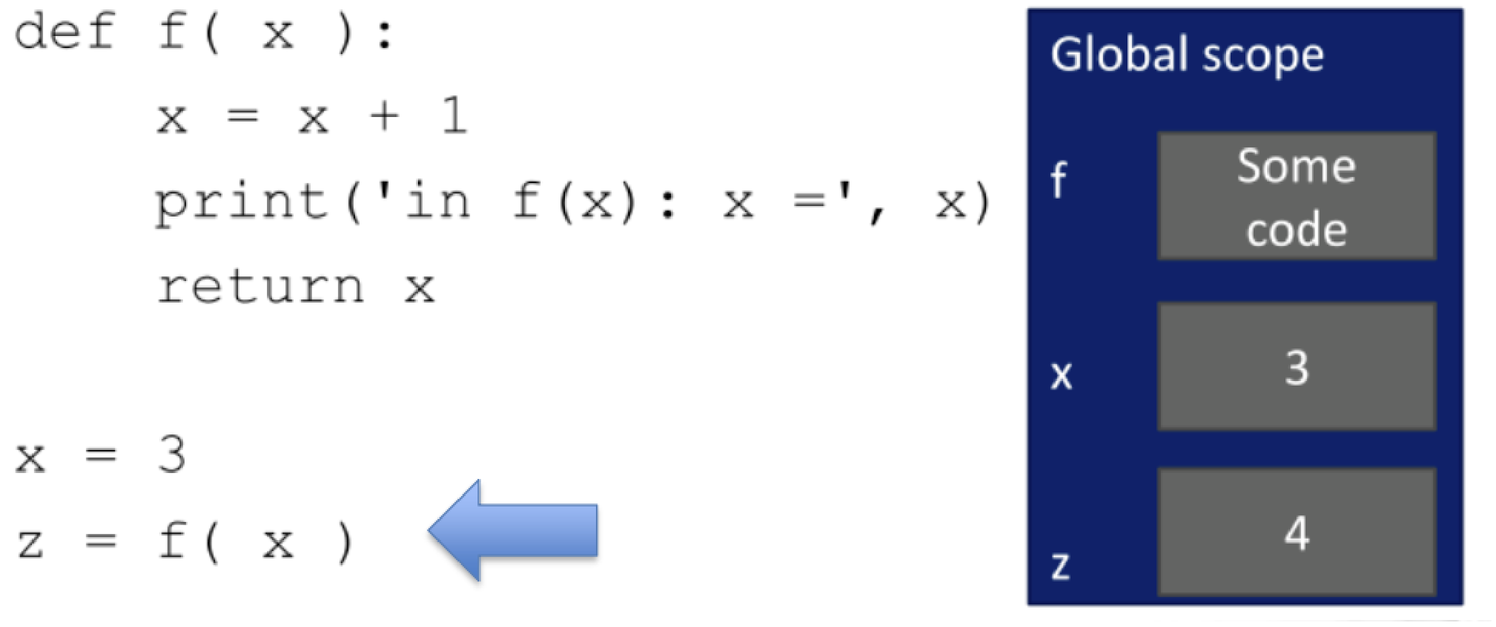
```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```



```
def f( x ):
    x = x + 1
    print('in f(x): x =', x)
    return x

x = 3
z = f( x )
```

- Inside a function, we can access a variable defined outside

```
from INSIDE fun2(): k:3
```

- If a variable name is defined twice, one as a global variable, and another is local to a function, inside the function, the local one will be used.

```
from INSIDE fun2(): k:5
```

- The code inside a function cannot modify a variable defined outside

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Input In [59], in <cell line: 5>()
      3     k+=2
      4     print(f"from INSIDE fun2(): k:{k}")
----> 5 fun2()

Input In [59], in fun2()
      2 def fun2():
----> 3     k+=2
      4     print(f"from INSIDE fun2(): k:{k}")

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

```
from INSIDE fun2(): k:5
from OUTSIDE fun2(): k:5
```

CS Cicles

- 10: def