# Data Science and Advanced Programming — Lecture 4

## Python Fundamentals II

Simon Scheidegger
Department of Economics, University of Lausanne, Switzerland

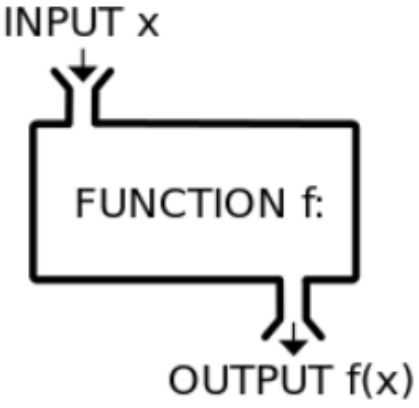
October 6th, 2025 | 12:30 - 16:00 | Internef 263

x

# Roadmap

1. Functions
2. Lists, Tuples, Dictionaries
3. Recursions

# 1. Functions



# How we learned to code so far

- **Until now:**
  - We have covered language mechanisms.
  - We know how to write different files for each computation.
  - We consider each file to be some piece of code.
  - We assume that each code is a sequence of instructions.
- **Problems with this approach:**
  - It is easy for small-scale problems.
  - However, it's messy for larger problems.
  - It's hard to keep track of details.
  - How do you know the right info is supplied to the right part of code?

:

# Good programming practice

- More code is not necessarily a good thing.
- Measure good programmers by the **amount of functionality**
  - → Introduce functions.
  - → Mechanism to achieve decomposition and abstraction.
  - → **Recycling**.

# Decomposition (generating a joint output)

# Functions

- In programming, it is useful to **divide the code** into modules that
  - are **self-contained**
  - used to break up code
  - intended to be **reusable**
  - keep code **organized**
  - keep code **coherent**
- **Today**: in this lecture, achieve decomposition with **functions**
- **Later**: achieve decomposition also with **classes**

# Functions → Abstractions

- In programming, think of a piece of code as a black box.
- do not (always) need to see details.
- do not (always) want to see details.
- hide tedious coding details.
  → achieve abstraction with function specifications or docstrings.

# The Purpose of Functions

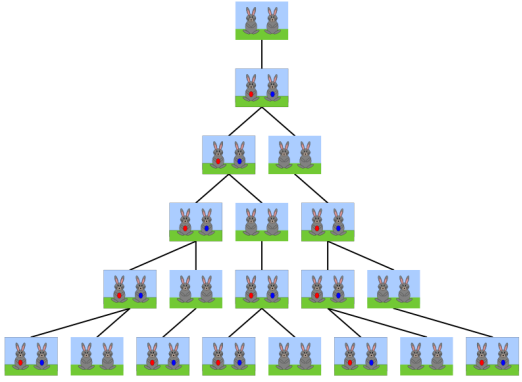
- write reusable pieces/chunks of code, called functions.
- **functions are not run in a program until they are "called'' or "invoked" in a program.**
- function characteristics:
  - has a name
  - has parameters (0 or more)
  - has a **docstring** (optional but recommended → allows re-use)
  - has a body
  - returns something

# Example: Fibonacci Sequence

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

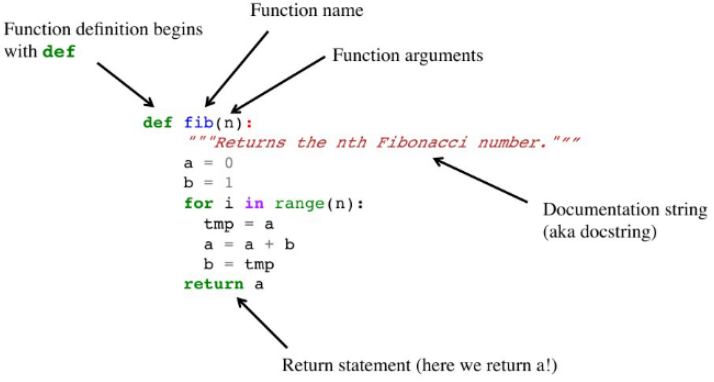
$$F_0 = 0, F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2},$$
$$\text{for } n > 1$$

One has $F_2 = 1$



# Functions: Definition I

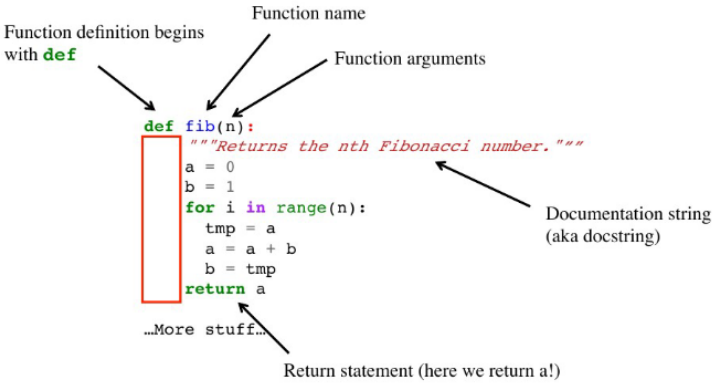
demo/example_1.py




```
def fib(n):
    """Returns the nth Fibonacci number."""
    a = 0
    b = 1
    for i in range(n):
        tmp = a
        a = a + b
        b = tmp
    return a
```

# Functions: Definition II

demo/example_1.py



# How to call Functions (from Terminal)

demo/example_1.py

```
>>> import example_1
('fib(5) =', 5)
>>> example_1.fib(3)
2
```

# How to call Functions

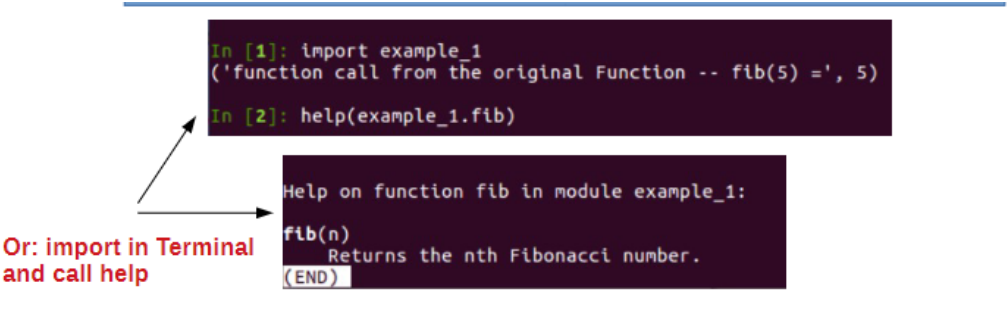
demo/example_1.py

```
def fib(n):
    """Returns the nth Fibonacci number."""
    a = 0
    b = 1
    for i in range(n):
      tmp = a
      a = a + b
      b = tmp
    return a
print("function call from the original Function -- fib(7) =",fib(7))
print(fib(10))
print(fib(20))
```

# How to call Functions from another File

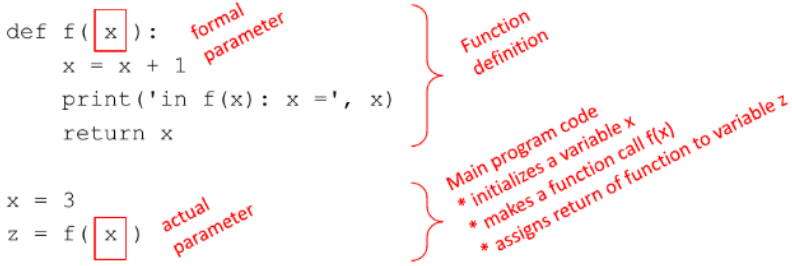
demo/example_1b.py

```python
#import the function to be present
import example_1 as ex
#call the function
print("Function call from another code: fib(5) =", ex.fib(5))
```



# Variable Scope

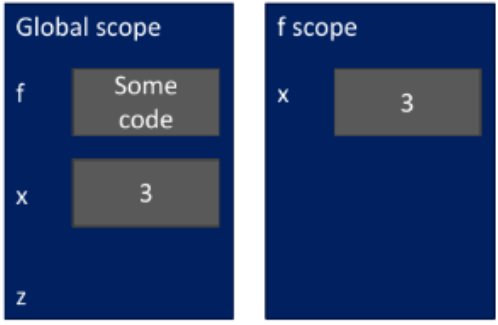
- **formal parameter** gets bound to the value of **actual parameter** when function is called
- **new scope**/frame/environment created when enter a function
- scope is mapping of names to objects



# Variable Scope II

See https://www.manning.com/books/get-programming

```
def f(x):
    x = x + 1
    print('in f(x): x =', x)
    return x
x = 3
z = f(x)
```



# Variable Scope III

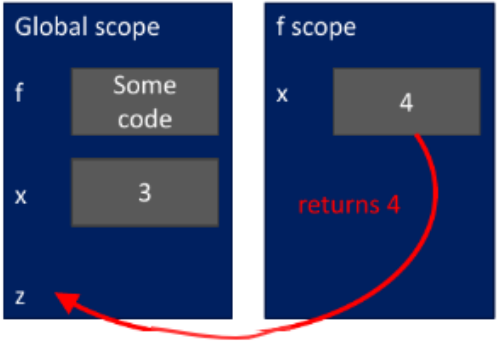
See https://www.manning.com/books/get-programming

```
def f(x):
    x = x + 1
    print('in f(x): x =', x)
    return x
x = 3
z = f(x)
```

# Variable Scope IV

See https://www.manning.com/books/get-programming

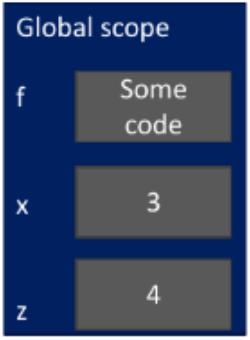
```
def f(x):
    x = x + 1
    print('in f(x): x =', x)
    return x
x = 3
z = f(x)
```



# Variable Scope V

demo/example_2.py

```
def f(x):
    x = x + 1
    print('in f(x): x =', x)
    return x
x = 3
z = f(x)
```



# More Scope Examples — shows accessing variables outside scope

demo/example_2b.py

```
def f(y):
    x = 1
    x += 1
    print(x)
x = 5
f(x)

def g(y):
    print(x)
    print(x+1)
x = 5
g(x)

def h(y):
    pass
    #x += 1 #leads to an error without line `global x` inside h
x = 5
h(x)
```

# More on Scope

demo/example_2c.py

```
def g(x):
    def h():
        x = 'abc'
        print(x)
    x = x + 1
    print('in g(x): x =', x)
    h()
    return x

x = 3
z = g(x)
```

# If still uncertain...

**!!!Go to Python Tutor!!!**

http://www.pythontutor.com/

# More Notes on Functions

- Functions always have a return value
  - Without a return statement the value is simply **None**
- Functions are objects
  - Can be used like any other object (list of functions, ...)
- Function arguments are passed by **assignment**
  - More precisely call by object reference
- No function overloading
  - Functions can't have the same name (even if the number of arguments differ)
- Operator overloading works, later...

# Once more: NO `return` STATEMENT

demo/example_3.py

```
def is_even_without_return( i ):
    """
    Input: i, a positive int
    Does not return anything
    """
    print('without return')
    remainder = i % 2

is_even_without_return(3)
```

- Python returns the value None, if no return given
- represents the absence of a value

# Once more: NO `return` STATEMENT

demo/example_3b.py

```
def is_even_with_return( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    print('with return')
    remainder = i % 2
    return remainder == 0

is_even_with_return(3)
print(is_even_with_return(4) )
```

- Python returns the value None, if no return given
- represents the absence of a value

# Recycling helps — short Code

demo/example_3b.py

```python
# Simple is_even function definition
def is_even( i ):
    """
    Input: i, a positive int
    Returns True if i is even, otherwise False
    """
    remainder = i % 2
    return remainder == 0

# Use the is_even function later on in the code
print("All numbers between 0 and 20: even or not")
for i in range(20):
    if is_even(i):
        print(i, "even")
    else:
        print(i, "odd")
```

# return vs print

## return

- return only has meaning inside a function
- only one return executed inside a function
- code inside function but after return statement not executed
- has a value associated with it, given to function caller

## print

- print can be used outside functions
- can execute many print statements inside a function
- code inside function can be executed after a print statement
- has a value associated with it, outputted to the console

# Functions: arguments such as functions

demo/example_4.py

```
def func_a():
    print('inside func_a')

def func_b(y):
    print('inside func_b')
    return y

def func_c(z):
    print('inside func_c')
    return z()

print(func_a())
print(5+func_b(2))
print(func_c(func_a))
```

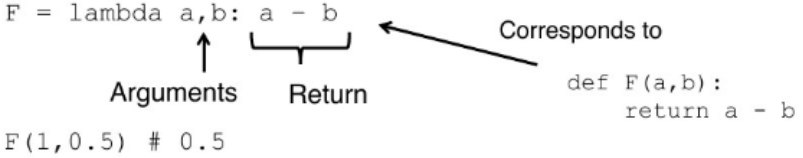
Arguments can take on any type, even functions

# Functions: lambda expressions

- Small anonymous functions can be created with the lambda keyword
- Example:

```
def apply(func,x):
    return func(x)
apply(lambda z: z ** 2)
```

- Convenient to define small functions



# Lambda Expressions

demo/example_6.py

```
def apply(func,x):
    return func(x)

x = apply(lambda z: z**2,2.)
print(x)
```

# Recall: Coding

Some very important conventions/rules:

- Use **4-space** indentation
- Wrap lines so that they don't exceed 79 characters
- Use blank lines to separate functions and classes, and larger blocks code inside functions
- **Use docstrings: documentation**
- No fancy encodings, even in comments!
- PEP8: `https://www.python.org/dev/peps/pep-0008/`
- Read it carefully...

# 2. Lists, Tuples, Dictionaries

# Compound data types

- So far, we have seen variable types: `int, float, bool,string.`
- We want now to introduce new **compound data types** (data types that are made up of other data types)
  - **Tuples** (similar to strings — sequences of something)
  - **Lists** (similar to strings — sequences of something)
- idea of aliasing
- idea of mutability
- idea of cloning

# Mutable vs. Immutable types

- **Mutable types**
  - Can change their contents / members
  - lists, dicts, user-defined types
- **Immutable types**
  - Cannot change their contents / members
  - most built-in types (`int`, `float`, `bool`, `str`, `tuple`)

# Lists

- Is an ordered sequence of information, accessible by index
- a list is denoted by **square brackets, [ ]**
- a list contains elements
- usually homogeneous (i.e., all integers)
- can contain mixed types (not common)
- list elements can be changed $\rightarrow$ **list is mutable**

# How to Use Lists

demo/example_9.py

```
# list is built-in
x = [0, 1, 2, 3, 3]

x[2] == 2 # access via [] index operator

x.insert(0, 5) # index, value
# x == [5, 0, 1, 2, 3, 3]

# remove by index - returns value
x.pop(0) # returns 5
# x == [0, 1, 2, 3, 3]

x = [0, 1, 2, 'three'] # can contain arbitrary types!

# access from the back with negative indices
x[-2] == 2
```

# More on Lists

```
# adding lists concatenates them
x += [4, 5, 6] # x == [0, 1, 2, 'three', 4, 5, 6]

# slicing [start:end + 1]
x[1:4] == [1, 2, 'three']

# slicing with a stride [start:end + 1:step]
x[0:7:2] == x[::2] == [0, 2, 4, 6]

# reverse slicing
x[-1:0:-2] == [6, 4, 2]
x[::-2] == [6, 4, 2, 0]
```

# Add Elements to a List

- add elements to end of list with `L.append(element)`
- mutates the list!

```
>>> L = [2, 1, 3]
>>> L.append(5)
>>> L
[2, 1, 3, 5]
```

- what is the dot?
  - lists are Python objects, everything in Python is an object
  - objects have data
  - objects have methods and functions

  access this information by `object_name.do_something()`

  - will learn more about these later very soon!

# Add Two or More Lists

demo/example_15a.py

- to combine lists together use concatenation, + operator, to give you a new list
- mutate list with `L.extend(some_list)`

```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2          # L3 is [2,1,3,4,5,6], L1, L2 unchanged
L1.extend([0,6])      # mutated L1 to [2,1,3,0,6]
```

# Remove Elements from a List

demo/example_15.py

- delete element at a specific index with `del(L[index])`
- remove a specific element with `L.remove(element)`
- looks for the element and removes it
- if element occurs multiple times, removes first occurrence
- if element not in list, gives an error

```
L = [2,1,3,6,3,7,0] # do below in order
L.remove(2)  # mutates L = [1,3,6,3,7,0]
L.remove(3)  # mutates L = [1,6,3,7,0]
del(L[1])    # mutates L = [1,3,7,0]
```

# Lists in the Memory

- Recall: **lists are mutable**
- They behave differently than immutable types
- is an object in memory
- variable name points to object
- **any variable pointing to that object is affected**

# How to Change Elements in a List

- Recall that lists are mutable!
- Assigning to an element at an index changes the value:

```
L = [2, 1, 3]
L[1] = 5
```

- `L` is now `[2,5,3]`, note this is the same object `L`

# Iterating over a List

- compute the sum of elements of a list
- common pattern, iterate over list elements

```
total = 0
for i in range(len(L)):
    total+=L[i]
    print(total)
```

```
total = 0
for i in L:
    total+=i
    print(total)
```

- Please note:
  - → list elements are indexed 0 to *len* $(L) - 1$
  - → range(n) goes from 0 to $\mathrm{n} - 1$

# Iterate over Lists

demo/example_14.py

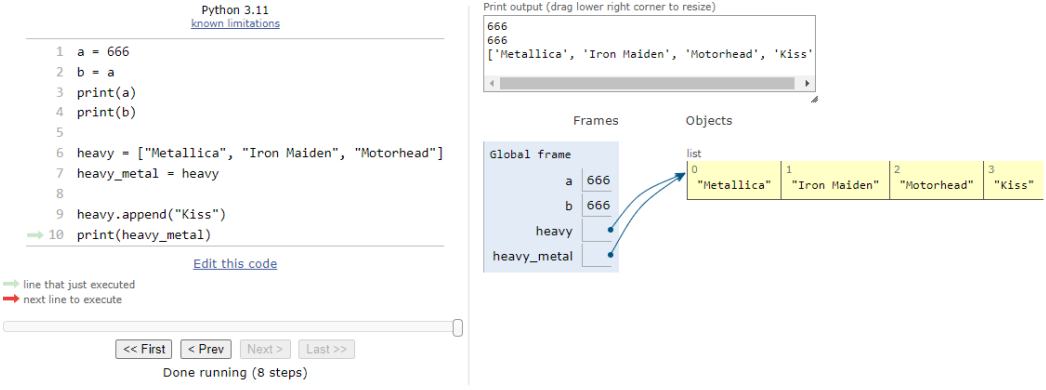
```
def sum_elem_method1(L):
  total = 0
  for i in range(len(L)):
      total += L[i]
  return total

def sum_elem_method2(L):
    total = 0
    for i in L:
        total += i
    return total

print(sum_elem_method1([1,2,3,4,5,6,7]))
print(sum_elem_method2([1,2,3,4]))
```

# http://www.pythontutor.com/

demo/example_16.py



# Cloning a List

demo/example_16.py

Python 3.11
known limitations

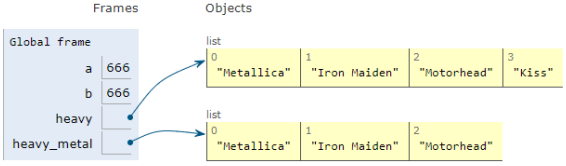
```
1  a = 666
2  b = a
3  print(a)
4  print(b)
5
6  heavy = ["Metallica", "Iron Maiden", "Motorhead"]
7  heavy_metal = heavy[:]
8
9  heavy.append("Kiss")
10 print(heavy_metal)
```

Edit this code

line that just executed
next line to execute

Print output (drag lower right corner to resize)

```
666
666
['Metallica', 'Iron Maiden', 'Motorhead']
```



# Cast Strings to Lists

demo/example_17.py

```
s = "I<3 cs"
print(list(s))
print(s.split('<'))
L = ['a', 'b', 'c']
print(''.join(L))
print('_'.join(L))
```

# Sort Lists

demo/example_18.py

```
L=[9,6,0,3]
print(sorted(L))
```

# More on Assignments

demo/example8.py

Be careful!

```
x = [0,1,2]
y = x
y[2] = 666
print(x) # [0, 1, 666]
print(y) # [0, 1, 666]
```

```
import copy
x = [0,1,2]
y = copy.copy(x)
y[2] = 666
print(x) # [0, 1, 2]
print(y) # [0, 1, 666]
```

Copy module for clean copies!

```
x = [0,1,[2,3]]
y = copy.copy(x)
y[2][0] = 666
print(x) # [0, 1, [666, 3]]
print(y) # [0, 1, [666, 3]]
```

Shallow copy

```
x = [0,1,[2,3]]
y = copy.deepcopy(x)
y[2][0] = 666
print(x) # [0, 1, [2  , 3]]
print(y) # [0, 1, [666, 3]]
```

Deep copy (usually what you want!)

# Passing Function Arguments — no return in function

- Consider

```
def incr(x):
    x += 1
x = 0
incr(x)
print(x)
```

- and

```
def incr_first(x):
    x[0] += 1
x = [0, 1, 2]
incr_first(x)
print(x)
```

- 0 — looks like pass by copy
- [1,1,2] — looks like pass by reference

# Passing Function Arguments — no return in function

- Consider

```
def incr(x):
    x += 1
x = 0
incr(x)
print(x)
```

- and

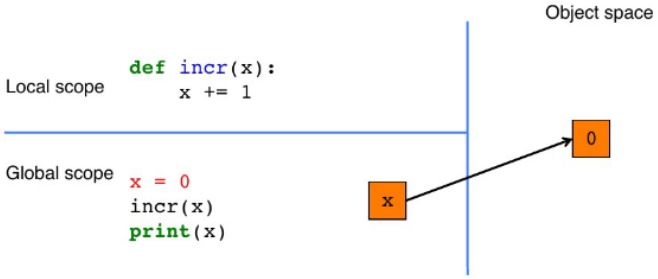
```
def incr_first(x):
    x[0] += 1
x = [0, 1, 2]
incr_first(x)
print(x)
```

- 0 — looks like pass by copy
- `[1,1,2]` — looks like pass by reference

# Passing function arguments: pass by assignment — no return in function

In Python:

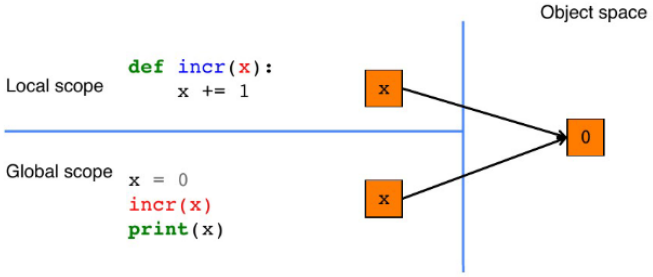
- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — no return in function

In Python:

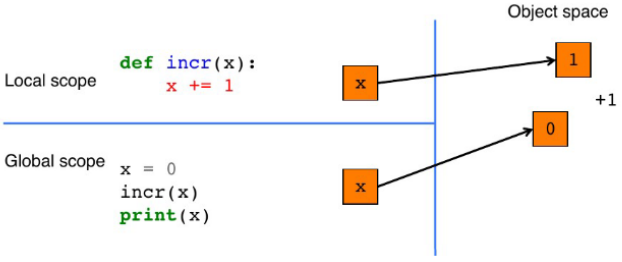
- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — no return in function

In Python:

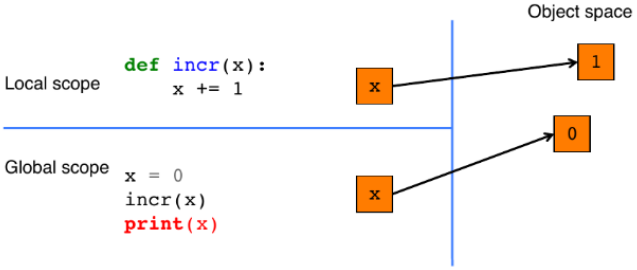
- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — no return in function

In Python:

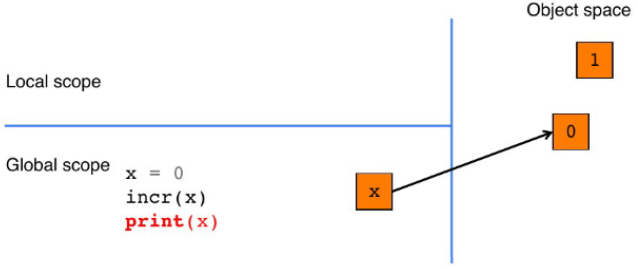
- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — no return in function

In Python:

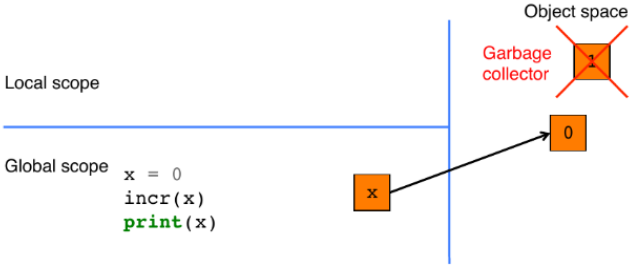
- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — no return in function

In Python:

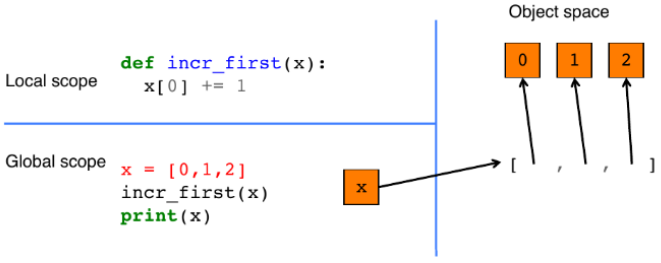
- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — no return in function

In Python:

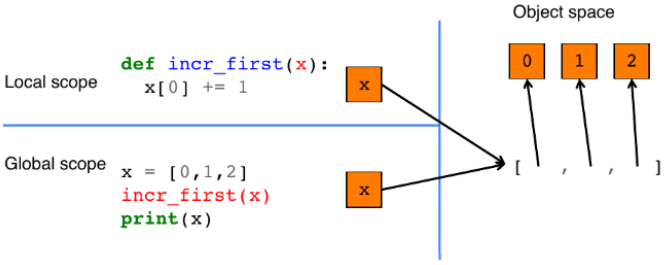
- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — no return in function

In Python:

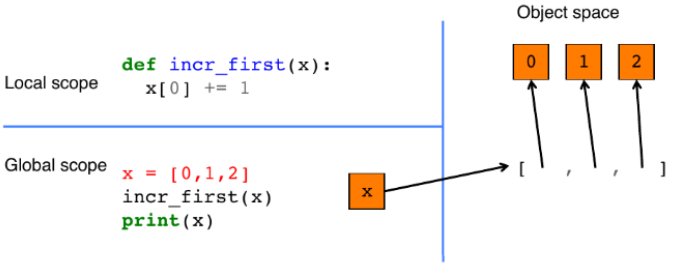
- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — no return in function

In Python:

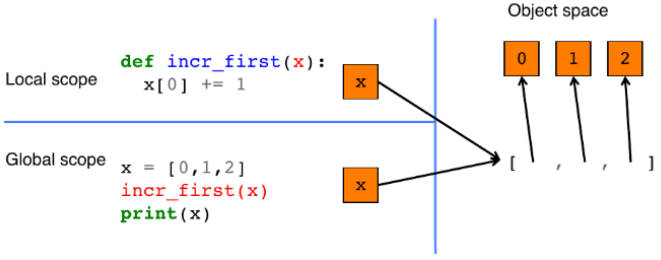
- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — no return in function

In Python:

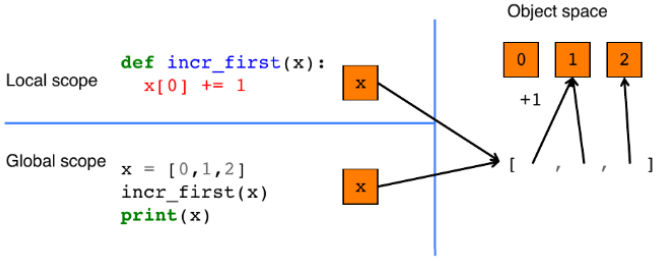
- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — no return in function

In Python:

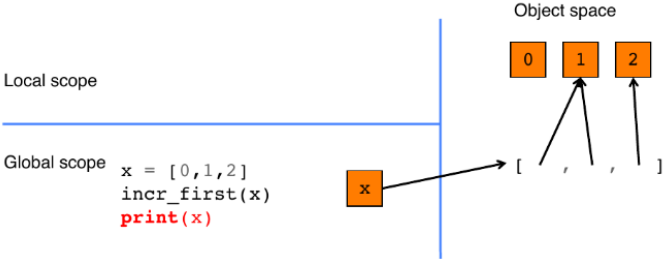
- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — no return in function

In Python:

- Variables are just names (labels)
- Names "bind" to an object when assigned to
- Assignment does not copy data



# Passing function arguments: pass by assignment — the code

demo/example_7.py

```
def incr(x):
    x += 1

x = 0
incr(x)
print(x)

def incr_first(x):
    x[0] += 1

x = [0,1,2]
incr_first(x)
print(x)
```

# More Built-in types: Tuples

- Tuples are more or less like a list but cannot be changed (i.e., are immutable)
- Use parentheses instead of brackets
- Support similar operations as lists

```
x=(1,2,3)
x[1]=3 # TypeError!
```

# Tuples

demo/example_10.py

Tuples are an ordered sequence of elements, can mix element types

```
# empty tuple
te = ()

t = (2,"HEC",3)
print(t[0])             #evaluates to 2

a=(2,"HEC",3) + (5,6) #  evaluates to (2,"HEC",3,5,6)
b = t[1:2]        #  slice tuple, evaluates to ("HEC",)
#Note: the extra comma means a tuple with 1 element
c = t[1:3]        #slice tuple, evaluates to ("HEC",3)
print(len(t))   #evaluates to 3
t[1] = 4      #gives an error, cannot modify object
```

# Potential usage of Tuples

demo/example_11.py

Conveniently used to swap more variables (first examples does not work, the second works):

```
x = y
y = x
```

```
temp = x
x = y
y = temp
```

```
(y, x) = (x, y)
```

Used to return more than one value from a function:

```
def quotient_and_remainder(x, y):
    q = x // y #integer division
    r = x % y
    return (q, r)
(quot, rem) = quotient_and_remainder(7,6)
```

# Iterate over Tuples

demo/example_13.py

```
>>> x = [(1,2), (3,4), (5,6)]
>>> for item in x:
... print "A tuple", item
A tuple (1, 2)
A tuple (3, 4)
A tuple (5, 6)
>>> for a, b in x:
... print "First", a, "then", b
First 1 then 2
First 3 then 4
First 5 then 6
```

# More Built-in types: Dictionaries

- If we want to manage a data collection of students, we can store information so far e.g., using separate lists for every info:

```
names = ['Tom', 'Keith', 'Marry', 'Megan']
grade = [6.0, ' 4.5,5.2,4.9]
course = ['Programming', 'Physics', 'Econometrics', 'Economics']
```

- A separate list for each item
- Each list must have the same length
- info stored across lists at same index, each index refers to info for a different person

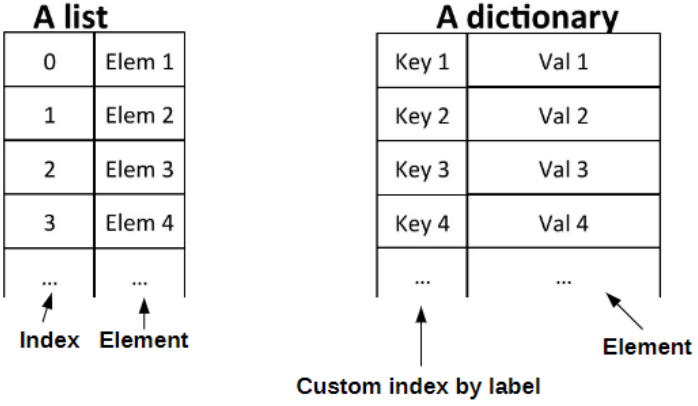
# Multiple Lists

```
def get_grade(student, name_list, grade_list, course_list):
    i = name_list.index(student)
    grade = grade_list[i]
    course = course_list[i]
    return (course, grade)
```

- **Messy** if have a lot of different info to keep track of
- **Must maintain many lists** and pass them as arguments
- Must always index using integers
- Must remember to change multiple lists

# Basic functionality of Dictionaries

- Nice to index item of interest directly (not always `int`)
- Nice to use one data structure, no separate lists



# A Dictionary in practice

- Store pairs of data
  - Key
  - Value
- similar to indexing into a list, looks up the key 'Keith'
- returns the value associated with Keith
- If key isn't found, you get an error

# Some operations on Dictionaries

demo/example_24.py

- **Add** an entry
  - `grades['Freddy'] = 4.9`
- **Test if key in dictionary**
  - `'Tom' in grades` → returns True
  - `'Daniel' in grades` → returns False
- **Delete** an entry
  - `del (grade ['Mary'])`

# Some operations on Dictionaries

demo/example_24.py

- Get an iterable that acts like a tuple of all keys
- `grades.keys()` → returns `['Mickey', 'Keith', 'Megan', 'Tom']`
- Get an iterable that acts like a tuple of all values
- `grades.values()` → returns `[5.0, 4.5, 4.9, 6.0]`
- Order is not guaranteed

# Dict — example

demo/example_19.py

```
x = dict(a=1, b=2, c='three')
x = {'a': 1, 'b': 2, 'c': 'three'}

# access via []
x['a'] == 1

# creating new entries
# any hashable type can be a key
x[1] = 4

# accessing keys, values or both
# order is not preserved
x.keys() # ['a', 'c', 1, 'b']
x.values() # [1, 'three', 4, 2]
x.items() # [('a', 1), (c, 'three'), (1, 4), ('b', 2)]
```

# Dictionary keys and values

demo/example_24.py

- **Values**
  - Any type (immutable and mutable)
  - Can be **duplicates**
  - Dictionary values can be lists, even other dictionaries!
- **Keys**
  - Must be **unique**
  - **Immutable** type (`int`, `float`, `string`, `tuple`, `bool`)
  - actually need an object that is **hash-table**, but think of as immutable as all immutable types are hash-table
  - Be careful with float type as a key
  - no order to keys or values!

```
d={4:{1: 0},(1,3): "twelve", 'const':[3.14,2.7,8.44]}
```

# `list` versus `dict`

## `list`

- **ordered** sequence of elements
- look up elements by an integer index
- indices have an **order**
- index is an **integer**

## `dict`

- **matches** “keys’‘ to “values”
- look up one item by another item
- **no order** is guaranteed
- key can be any **immutable** type

# Dictionaries — summary

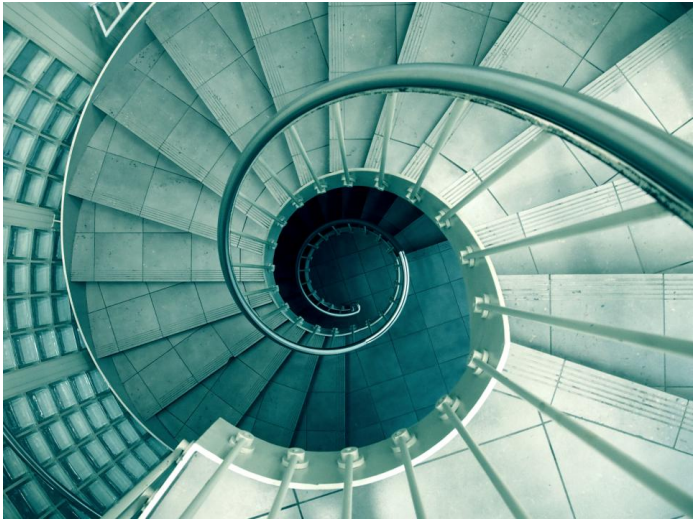
```
x = dict(a=1, b=2, c='three')
x = {'a': 1, 'b': 2, 'c': 'three'}

# access via []
x['a'] == 1

# creating new entries
# any hashable type can be a key
x[1] = 4

# accessing keys, values or both
# order is not preserved
x.keys() # ['a', 'c', 1, 'b']
x.values() # [1, 'three', 4, 2]
x.items() # [('a', 1), (c, 'three'), (1, 4), ('b', 2)]
```

# 3. Recursion



# Recursion

- **Algorithmically**: a way to design solutions to problems by divide-and-conquer or decrease-and-conquer
  - Reduce a problem to simpler versions of the same problem.
- **Semantically**: a programming technique where a function calls itself
  - In programming, the goal is to NOT have infinite recursion
    - Must have 1 or more **base cases** that are easy to solve
    - Must solve the same problem on some other input with the goal of simplifying the larger problem input.

# Iterative algorithms (for loops)

demo/example_20.py

- Looping constructs (`while` and `for` loops) lead to iterative algorithms.
- Can capture computation in a set of state variables that update on each iteration through loop.

```
def mult_a(a, b):
    result = 0
    while b > 0:
        result += a
        b -= 1
    return result

print(mult_a(2,10))
```

"multiply a*b" is equivalent to "add a to itself b times"
$\rightarrow a + a + a + a + \ldots + a$

# The alternative — a recursive way

demo/example_21.py

demo/example_22.py

The recursive step: *Think how to reduce the problem to a simpler/smaller version of same problem*

- Base case
- Keep reducing the problem until we reach a simple case that can be solved directly
- when $b = 1, a * b = a$

$$
\begin{aligned}
a * b &= a + a + a + a + \ldots + a \quad \text{(n times)a} \\
&= a + a + a + a + \ldots + a \quad (1 + n - 1 \text{ times})a \\
&= a + a * (b - 1) \quad \text{recursive reduction}
\end{aligned}
$$

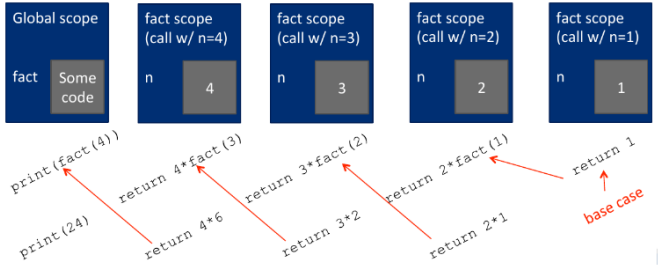
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n*factorial(n-1)
print(factorial(10))
```

```
def mult_iter(a, b):
    if b == 1:
        return a
    else:
        return a + mult_iter(a, b-1)
print mult_iter(1,10)
```

# What's going on in factorial?

```
def fact(n):
    if n == 1:
        return 1
    else:
        return n*fact(n-1)

print(fact(4))
```



# Iteration versus Recursion

```
def factorial_iter(n) :
    prod = 1
    for i in range(1, n+1):
        prod *= i
    return prod
```

```
def factorial(n):
    if n==1:
        return 1
    else:
        return n*factorial(n-1)
```

- recursion may be simpler, more intuitive
- recursion may be efficient from programmer's point of view
- recursion may not be efficient from a computer point of view

# A famous Example: The Fibonacci sequence

- Leonardo of Pisa (aka Fibonacci) modeled the following challenge.
- Newborn pairs of rabbits (one female, one male) are put in a pen.
- Rabbits mate at the age of one month.
- Rabbits have a one-month gestation period.
- Assume rabbits never die, and that female always produces one new pair (one male, one female) every month from its second month on.
- $\rightarrow$ **How many female rabbits are there at the end of one year?**

# Fibonacci Sequence — the code

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

```
def F(n):
    if n == 0: return 0
    elif n == 1: return 1
    else: return F(n-1) + F(n-2)
```

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

```
startNumber = int(input("Enter the start number here "))
endNumber = int(input("Enter the end number here "))
```

# Questions?