# SciComp with Py

# Functions & Sequences

Vladimir Kulyukin
Department of Computer Science
Utah State University

# Outline

- 2nd (and Firmer!) Handshake with Functions
- Sequences: Strings, Lists, Tuples (aka Toops)
- Indexing & Slicing Sequences
- Iterating over Sequences
- Destructive & Constructive Modification of Sequences
- Function References
- Anonymous Functions

# 2nd Handshake with Functions

# Newton Square Root Approximation

# Newton's Square Root Approximation

- A common way to compute the square root of a number **n** is to use Newton's method
- Suppose that we initially guess that **sqrt(n) =** $\mathbf{g_0}$; a better guess $\mathbf{g_1}$ is obtained by computing the average of $\mathbf{g_0}$ and $\mathbf{n/g_0}$
- In general, given the current guess $\mathbf{g_i}$, the next guess $\mathbf{g_{i+1}}$ is obtained as $\mathbf{(g_i + n/g_i)/2}$
- We keep going until our current guess is good enough

# Functional Abstraction

We can abstract the following functions from the algorithm's description to implement it:

1) **average(x, y)**: average of **x** and **y**
2) **next_guess(n, g)**: next guess to **sqrt(n)** given current guess **g**
3) **is_good_enough(n, g, error)**: is guess **g** good enough?
4) **newton_sqrt_root(n, g, error)**: keep going until **g** is good enough
5) **newton_sqrt(n)**: call **newton_sqrt_root(n, 1, 0.0001)**

# Py2 Solution

# AVERAGE

```
def average(x, y):
    return (x+y)/2.0
```

But keep in mind that you can also do

**from __future__ import division**

# NEXT_GUESS

This is next guess

This is current guess

Math

$$g_{i+1} = \frac{g_i + \frac{n}{g_i}}{2}$$

Py

```
def next_guess(n, g):
    return average(g, float(n)/g)
```

# IS_GOOD_ENOUGH

Math $|g^2 - n| \leq \epsilon$

Py

```
def is_good_enough(n, g, error):
    return abs(g**2 – n) <= error
```

# NEWTON_SQUARE_ROOT

```
def newton_square_root(n, g, error):
    if is_good_enough(n, g, error):
        return g
    else:
        ng = next_guess(n, g)
        return newton_square_root(n, ng, error)
```

# Py Solution: NEWTON_SQRT

**0.0001 is the default value of error**

```
def newton_sqrt(n):
    return newton_square_root(n, 1, 0.0001)
```

# Py2 & Py3 Solutions Side by Side

Py2 solution is in newton_square_root.py

Py3 solution is in newton_square_root_py3.py

# Py Sequences

# Strings, Lists, Tuples

# Some Terminology

- **BUILT-IN** - sequence type provided as part of the language
- **HOMOGENEOUS** – sequence can store elements of only one type
- **HETEROGENEOUS** – sequence can store elements of different types
- **INDEXED/INDEXABLE** – elements of a sequence can be referred to by integers
- **MUTABLE** – any element at a legal index can be changed
- **IMMUTABLE** – no element at a legal index can be changed

# Strings

- Strings are **built-in**, **homogeneous**, **indexed**, & **immutable** sequences of characters enclosed in matching double or single quotes
- Examples: **“one”, 'two', “abracadabra”, 'abracadabra'**
- String indexing is **0**-based
- Strings are immutable: cannot be assigned into, only read out of

# Example

```
>>> s = 'abracadabra'
>>> s[0]
'a'
>>> s[2]
'r'
>>> s[0] = 'o'

Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    s[0] = 'o'
TypeError: 'str' object does not support item assignment
```

**This error shows that strings are immutable**

# Lists

- Lists are **built-in**, **heterogeneous**, **indexed**, & **mutable** sequences
- A list is any sequence of valid Py elements separated by commas and enclosed in **[ ]**
- Lists are heterogeneous: **[1, 2, 3], [1, 'one', 2, “two”]**
- List indexing is **0**-based, i.e., indexes start at **0**
- A list can be assigned to a regular variable (so long as the variable's name is legal)
- Examples:

**lst0 = [1, 2, 3]**
**lst1 = [1, 'one', 2, 'two']**

# Programmatic Construction of Lists

Lists can be programmatically constructed by the methods **list.insert()** and **list.append()**; for those of you who know OOP, note that lists are objects:

**list.insert(i, x)** inserts **x** at position i in the list
**list.append(x)** inserts **x** at the end of the list

# Example: Adding Elements on the Left

## Py Source

```
lstx = [ ]
lstx.insert(0, 3)
lstx.insert(0, 2)
lstx.insert(0, 1)

print 'lstx[0]', '=', lstx[0]
print 'lstx[1]', '=', lstx[1]
print 'lstx[2]', '=', lstx[2]
```

## Output

```
lstx[0] = 1
lstx[1] = 2
lstx[2] = 3
```

# Function LIST()

Py has **list()** function that can be used as a constructor to make empty lists or convert other sequences to lists; comes in handy when you need to convert strings to lists of characters

```
>>> x = list()
>>> x
[ ]
>>> y = list("abc")
>>> y
['a', 'b', 'c']
```

# Tuples

- Tuples are **built-in**, **heterogeneous**, **indexed**, & **immutable** sequences of elements enclosed in **( )**
- Examples: **(1, 2, 3)**, **(1, “one”, 2, 'two')**
- Indexing is **0**-based
- Tuples are immutable: cannot be assigned into, only read out of

# Example: Immutability of Tuples

```
>>> t = (1, 'a', 2)
>>> t[0] = 10

Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    t[0] = 10
TypeError: 'tuple' object does not support item assignment
```

**This error shows that tuples are immutable**

# Indexing & Slicing

# Non-Negative Indexing: Lists

Non-negative indexing allows you to access elements in lists *from left to right*. Examples:

```
lst = [1, 2, 3]
lst[0] == 1 ## returns True
lst[1] == 2 ## returns True
lst[2] == 3 ## returns True
lst[4] ## Index out of range error
```

# Non-Negative Indexing: Tuples

Non-negative indexing allows us to access elements in tuples *from left to right*. Examples:

```
toop = (1, 2, 3)
toop[0] == 1 ## returns True
toop[1] == 2 ## returns True
toop[2] == 3 ## returns True
toop[4] ## Index out of range error
```

# Non-Negative Indexing: Strings

Non-negative indexing allows us to access elements in strings *from left to right*. Examples:

```
s = 'abracadabra'
s[0] == 'a'  ## returns True
s[1] == 'b' ## returns True
s[2] == 'r'  ## returns True
s[2] == 'w' ## returns False
```

# Negative Indexing

Negative indexing allows us to access elements in sequences *from right to left*. Negative indexing works the same way on lists, toops, strings. Examples:

```
lst = ['a', 'b', 'c']
lst[-1] == 'c' ## returns True, because 'a' is 1st element from the right
lst[-2] == 'b' ## returns True, because 'b' is 2nd element from the right
lst[-3] == 'a' ## returns True, because 'c' is 3rd element from the right
```

# Sequence Slicing

- If **seq** is a sequence (list, string, toop), a **slice** of that **seq** is a sub-sequence of elements
- For example, if **seq** is a list, then a slice is a sub-list; if **seq** is a tuple, then a slice is a tuple
- A slice is specified by its start and end indexes; e.g., **seq[start:end]**
- A slice **seq[start:end]** includes all elements from **seq[start]** upto **seq[end-1]**

# Sequence Slicing

Both negative and non-negative indices can be used in slicing. Examples:

```
lst = ['one', 'two', 'three', 'four', 'five']
slice_01 = lst[1:4] ## ['two', 'three', 'four']
lst = [1, 2, 3, 4, 5]
lst[-4:-1] == [2, 3, 4] ## returns True
## lst[-4:-1] consists of lst[-4], lst[-3], and lst[-2]. The
## element lst[-1] is not included.
```

# Sequence Slicing

- If the start index of a slice is omitted, the slice is assumed to start at 0
- If the end index of a slice is omitted, the slice is assumed to go upto and *including* the last valid index
- Examples:

```
lst = [1, 2, 3, 4, 5]
lst[:3] == [1, 2, 3] ## returns True
lst = [1, 2, 3, 4, 5]
print lst[2:], "\n" ## prints [3, 4, 5]
```

# Stepping Through Slices

- You can specify a sampling step when specifying a slice
- Both non-negative and negative indices can be used
- Examples:

```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
lst[0:10:3]     ## every 3rd element from 0 upto 9: [1, 4, 7, 10]
lst[1:10:2]     ## every 2nd element from 1 upto 9: [2, 4, 6, 8, 10]
lst[9:0:-3]     ## every 3rd element from 9 to 1: [10, 7, 4]
lst[5::-2]      ## start at lst[5] and go left every 2nd element
lst[:5:-2]      ## end at 5 by going left from the right end of the list
```

# Iterating Over Sequences

# Iteration

- Sequence iteration is an operation of traversing individual elements of a sequence in a specific order, typically, either left to right or right to left
- Standard control structures are used to iterate over sequences: for & while loops
- Sequences can also be iterated over with list comprehension and generators (we will study both later in the course)

# Problem

Write a Py program that take a list of numbers and prints each number on a separate line.

# FOR Loop

## Py Source

```
lst = [1, 2, 3, 4]

for x in lst:

    print x
```

## Output

```
1
2
3
4
```

# FOR Loops & Ranges

Py Source

```
lst = [1, 2, 3, 4]

for i in xrange(0, len(lst)):

    print 'lst[', i, ']=', lst[i]
```

Output

```
lst[ 0 ]= 1
lst[ 1 ]= 2
lst[ 2 ]= 3
lst[ 3 ]= 4
```

# Numerical Ranges in Py2 & Py3

- Py2 has two numerical range functions: **range(start, end)** & **xrange(start, end)**

  **range()** constructs an explicit list from **start** to **end-1**

  **xrange()** constructs a lazy sequence from **start** to **end-1**

  in lazy sequences, elements are produced on demand, one at a time, as they are accessed, w/o storing all elements in memory

  bottom line: in Py2, use **xrange()** on large ranges, e.g., **xrange(0, 1000000)**

- Py3 does not have **xrange(); range()** works the same way as **xrange()** in Py2: this is another case of backward incompatibility

# Numerical Ranges in Py2 & Py3

## Py2

```
lst = [1, 2, 3, 4]
for i in xrange(0, len(lst)):
    print 'lst[', i, ']=', lst[i]
```

## Py3

```
lst = [1, 2, 3, 4]
for i in range(0, len(lst)):
    print('lst[', i, ']=', lst[i])
```

## Py2 Output

```
lst[ 0 ]= 1
lst[ 1 ]= 2
lst[ 2 ]= 3
lst[ 3 ]= 4
```

## Py3 Output

```
lst[ 0 ]= 1
lst[ 1 ]= 2
lst[ 2 ]= 3
lst[ 3 ]= 4
```

# Problem

Write Py & PL programs that take a list of numbers and replaces each number in the list with its square root.

# List Iteration & Destructive Modification

Py2 Source

```
lst = [1, 2, 3, 4]
for i in xrange(0, len(lst)):
    lst[i] **= .5
for x in lst: print x
```

Output

```
1.0
1.41421356237
1.73205080757
2.0
```

# Destructive Modification of Sequences

# Passing Sequences by Reference

- In Py, sequences are passed to functions *by reference*, not by value (there is no such thing in Py as *pass by value* like in C++)
- If a function destructively modifies a mutable sequence, e.g., a list or a set, passed to it, the sequence is permanently modified
- If you want to keep the original sequence unmodified, make a copy of it

# Passing Lists by Reference

## Problem

Write a function that takes a list of numbers and destructively modifies it by replacing each number with its square root. This is called *destructive modification*.

Py2 source is in py2_lists.py
Py3 source is in py3_lists.py

# Passing Lists by Reference

## Py2

```
def sqrt_list(a_lst):
    for i in xrange(0, len(a_lst)):
        a_lst[i] **= .5

lst = [1, 2, 3, 4]
print lst
sqrt_list(lst)
print lst
```

## Py2 Output

```
[1, 2, 3, 4]
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
```

# Passing Lists by Reference

## Py3

```
def sqrt_list(a_lst):
    for i in range(0, len(a_lst)):
        a_lst[i] **= .5

lst = [1, 2, 3, 4]
print(lst)
sqrt_list(lst)
print(lst)
```

## Py3 Output

```
[1, 2, 3, 4]
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
```

# Passing Sets by Reference

Problem

Write a function that takes a set of numbers and destructively modifies it by removing a given number if that number is in the set.

Py2 source is in py2_sets.py
Py3 source is in py3_sets.py

# Passing Sets by Reference in Py2

Py2

```
def remove_from_set(x, a_set):
    if x in a_set:
        a_set.remove(x)


s = set((1, 2, 3, 4))
print s
remove_from_set(3, s)
print s
remove_from_set(10, s)
print s
```

Py2 Output

```
set([1, 2, 3, 4])

set([1, 2, 4])

set([1, 2, 4])
```

# Passing Sets by Reference in Py3

## Py3

```
def remove_from_set(x, a_set):
    if x in a_set:
        a_set.remove(x)


s = set((1, 2, 3, 4))
print(s)
remove_from_set(3, s)
print(s)
remove_from_set(10, s)
print(s)
```

## Py3 Output

Note that sets in Py3 are displayed differently than in Py2

```
{1, 2, 3, 4}
{1, 2, 4}
{1, 2, 4}
```

# Constructive Modification of Sequences

# Copying Lists

## Problem

Write a function that takes a list of numbers and returns a new list of the square roots of the numbers in the original list. The original list is unchanged.

Py2 source is in py2_lists.py
Py3 source is in py3_lists.py

# Copying Lists

Making a shallow copy of argument list

## Py2

```
import copy

def sqrt_list_copy(a_list):
    shallow_copy = copy.copy(a_list)
    for i in xrange(len(shallow_copy)):
        shallow_copy[i] **= 0.5
    return shallow_copy

lst = [1, 2, 3, 4]
print lst
sqrt_of_lst = sqrt_list_copy(lst)
print sqrt_of_lst
print lst
```

## Py2 Output

```
[1, 2, 3, 4]
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
[1, 2, 3, 4]
```

# Copying Lists

Making a shallow copy of argument list

## Py3

```
import copy

def sqrt_list_copy(a_list):
    shallow_copy = copy.copy(a_list)
    for i in range(len(a_list)):
        shallow_copy[i] **= 0.5
    return shallow_copy

lst = [1, 2, 3, 4]
print(lst)
sqrt_of_lst = sqrt_list_copy(lst)
print(sqrt_of_lst)
print(lst)
```

## Py3 Output

```
[1, 2, 3, 4]
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0]
[1, 2, 3, 4]
```

# Copying Sets

Problem

Write a function that takes a set ***S*** of numbers and a number ***x*** and returns a new set that contains all numbers of ***S*** except ***x***. The original set is unchanged.

Py2 source is in py2_sets.py
Py3 source is in py3_sets.py

# Copying Sets in Py2

## Py2

```
def remove_from_set_copy(x, a_set):
    shallow_copy = a_set.copy()
    if x in a_set:
        shallow_copy.remove(x)
        return shallow_copy
    else:
        return shallow_copy

s = set([1, 2, 3, 4])
print s
copy_of_s = remove_from_set_copy(2, s)
print copy_of_s
print s
```

## Py2 Output

```
set([1, 2, 3, 4])
set([1, 3, 4])
set([1, 2, 3, 4])]
```

# Copying Sets in Py3

## Py3

```
def remove_from_set_copy(x, a_set):
    shallow_copy = a_set.copy()
    if x in a_set:
        shallow_copy.remove(x)
        return shallow_copy
    else:
        return shallow_copy

s = set([1, 2, 3, 4])
print(s)
copy_of_s = remove_from_set_copy(2, s)
print(copy_of_s)
print(s)
```

## Py3 Output

```
{1, 2, 3, 4}
{1, 3, 4}
{1, 2, 3, 4}
```

# Function References

# Function References

- It is possible to pass functions by reference to other functions/methods
- Functions are first-order objects, i.e., there is nothing special about a function reference - once there is a name for a function, it is possible to pass that name to other functions
- If a function has a name, you can pass a named reference
- If a function does not have a name, you can pass an anonymous reference

# Passing Named Function References

## Problem

Write a function predicate that returns **True** if a number is prime and **False** when it is not. Then write another function that takes a reference to this predicate to compute a list of primes in a specific range.

Py2 source is in py2_prime_sifter.py
Py3 source is in py3_prime_sifter.py

# Two Versions of IS_PRIME Predicate in Py2

```
import math

def is_prime(n):
    if n < 2: return False
    if n == 2: return True
    for d in xrange(2, int(math.sqrt(n))+1):
        if n % d == 0:
            return False
    return True
```

```
import math

def is_prime2(n):
    if n < 2: return False
    if n == 2: return True
    for d in xrange(2, int(n/2.0)+1):
        if n % d == 0:
            return False
    return True
```

# Two Versions of IS_PRIME Predicate in Py3

```
import math

def is_prime(n):
    if n < 2: return False
    if n == 2: return True
    for d in range(2, int(math.sqrt(n))+1):
        if n % d == 0:
            return False
    return True
```

```
import math

def is_prime2(n):
    if n < 2: return False
    if n == 2: return True
    for d in range(2, int(n/2)+1):
        if n % d == 0:
            return False
    return True
```

# Passing Named Function Reference to Another Function

## Py2

```
def sift_primes_in_range(prime_pred, number_range):
    prime_list = []
    for n in number_range:
        if prime_pred(n):
            prime_list.append(n)
    return prime_list


prime_list_1 = sift_primes_in_range(is_prime, xrange(0, 31))
print prime_list_1
prime_list_2 = sift_primes_in_range(is_prime2, xrange(0, 31))
print prime_list_2
```

is_prime is a named function reference

## Py2 Output

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

is_prime2 is a named function reference

# Passing Named Function Reference to Another Function

## Py3

```
def sift_primes_in_range(prime_pred, number_range):
    prime_list = []
    for n in number_range:
        if prime_pred(n):
            prime_list.append(n)
    return prime_list

prime_list_1 = sift_primes_in_range(is_prime, range(0, 31))
print(prime_list_1)
prime_list_2 = sift_primes_in_range(is_prime2, range(0, 31))
print(prime_list_2)
```

is_prime is a named function referencev

is_prime2 is a named function reference

## Py3 Output

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

# Anonymous Functions

# Py Lambdas

- Sometimes we may not want to define a named function
- Instead we would like to create a function as our program is running and then throw the function away instead of formally defining it with a name
- These forms are known as **lambdas** (aka anonymous functions)

# Py Lambdas

- Lambda functions were planned to be dropped in Py3
- The Py3 community's arguments was as follows: lambdas are not needed when there is list comprehension (more on list comprehension later in the course)
- Having both lambdas & list comprehension violates the *Zen of Python*: there should be one obvious way to solve the problem
- However, lambdas are still part of Py3 due to the fierce resistance from the Py2 community

# Py Lambda Syntax

- The general syntax for the lambda expression in Py is as follows: **lambda argument_list: expression**
- **argument_list** is a comma-separated list of arguments
- **expression** is any valid Py expression
- Examples:
  - **lambda x: x**2**
  - **lambda x, y: x**2 + y**3**
  - **lambda x, y, z: (x**2, y**3, z**4)**

# Mapping Functions Over Sequences

- Anonymous functions are used in mapping over sequences
- In Py2 & Py3, the syntax is the same:

  **map(fun, seq0, seq1, seq2, …, seqn)**
- The number of sequences, i.e., **n**, depends on the number of arguments of the function **fun**; this function can be named or anonymous
- In Py2, map returns a list of outputs of applying fun to the corresponding elements of **seq0**, **seq1**, **seq2**, …, **seqn**
- In Py3, map returns an *iterable* map object

# Mapping Named Functions Over Sequences in Py2

## Named Py2 Function

```
def add10(x):
    return x+10
```

## Mapping Named Py2 Function

```
>>> map(add10, [1, 2, 3, 4])
[11, 12, 13, 14]
>>> map(add10, (1, 2, 3, 4))
[11, 12, 13, 14]
>>> map(add10, set([1, 2, 3, 4]))
[11, 12, 13, 14]
```

Mapping add10 over a list

Mapping add10 over a tuple

Mapping add10 over a set

# Mapping Named Functions Over Sequences in Py2

## Named Py2 Function

```
def add10(x):
    return x+10
```

## Mapping Anonymous Py2 Function

```
>>> map(lambda x: x+1, [1, 2, 3])
[2, 3, 4]
>>> map(lambda x: x*10, (1, 2, 3))
[10, 20, 30]
>>> map(lambda c: ord(c), set('abc'))
[97, 99, 98]
>>> map(lambda x, y: x*y, [1, 2, 3], (4, 5, 6))
[4, 10, 18]
```

Mapping lamda over a list

Mapping lamda over a tuple

Mapping lamda over a set

Mapping 2-argument lambda over a list and a tuple

# Mapping Named Functions Over Sequences in Py2

Named Py2 Function

```
def f(x, y):
    return x*y
```

Mapping Named Two-Argument Py2 Function

```
>>> map(f, [1, 2, 3], [4, 5, 6])
[4, 10, 18]
>>> map(f, (1, 2, 3), [4, 5, 6])
[4, 10, 18]
>>> map(f, set([1, 2, 3]), [4, 5, 6])
[4, 10, 18]
```

# Mapping Anonymous Functions Over Sequences in Py2

```
>>> map(lambda x: x+1, [1, 2, 3])
[2, 3, 4]
>>> map(lambda x: x*10, (1, 2, 3))
[10, 20, 30]
>>> map(lambda c: ord(c), set('abc'))
[97, 99, 98]
>>> map(lambda x, y: x*y, [1, 2, 3], (4, 5, 6))
[4, 10, 18]
```

Note that in Py2 map operations always return lists

# Mapping Named Functions Over Sequences in Py3

## Named Py3 Function

```
def add10(x):
    return x+10
```

## Mapping Named Py3 Function

```
>>> map1 = map(add10, [1, 2, 3, 4])
>>> map2 = map(add10, (1, 2, 3, 4))
>>> map3 = map(add10, set([1, 2, 3, 4]))
>>> map1
<map object at 0x7f1d60d40400>
>>> map2
<map object at 0x7f1d60d40b70>
>>> map3
<map object at 0x7f1d60d40b00>
```

In Py3 mapping operations return map objects

# Iterating over Maps in Py3

## Py3 Map

```
>>> map1 = map(add10, [1, 2, 3, 4])
>>> map1
<map object at 0x7f1d60d40400>
```

## Iterating over Py3 Map

```
>>> for i in map1:
        print(i)
11
12
13
14
```

# List Construction from Maps in Py3

## Py3 Map

```
>>> map2 = map(add10, (1, 2, 3, 4))
>>> map2
<map object at 0x7f1d60d40b70>
```

## List Construction

```
>>> lst = list(map2)
>>> lst
[11, 12, 13, 14]
>>> s = set(map2)
>>> s
set()
```

You can iterate over a map only once

# Tuple Construction from Maps in Py3

## Py3 Map

```
>>> map3 = map(add10, set([1, 2, 3, 4]))
>>> map3
<map object at 0x7f1d60d40b00>
```

## Tuple Construction

```
>>> t = tuple(map3)
>>> t
(11, 12, 13, 14)
```

# Sequence Construction with Maps & Lambdas in Py3

```
>>> set(map(lambda c: ord(c), 'abc'))
{97, 98, 99}
>>> list(map(lambda x, y: x+y, [1, 2, 3], [4, 5, 6]))
[5, 7, 9]
```

# References

- Here is a quick reference on Newton's method of computing square roots:http://mathworld.wolfram.com/NewtonsIteration.html