Python Programming for Data Processing and Climate Analysis

Jules Kouatchou and Hamid Oloso

Jules.Kouatchou@nasa.gov and Amidu.o.Oloso@nasa.gov

Goddard Space Flight Center
Software System Support Office
Code 610.3

February 25, 2013

Training Objectives

We want to introduce:

- Basic concepts of Python programming
- Array manipulations
- Handling of files
- 2D visualization
- EOFs

Obtaining the Material

Slides for this session of the training are available from:

https://modelingguru.nasa.gov/docs/DOC-2315

You can obtain materials presented here on *discover* at

/discover/nobackup/jkouatch/pythonTrainingGSFC.tar.gz

After you untar the above file, you will obtain the directory *pythonTrainingGSFC/* that contains:

```
Examples/
Slides/
```

Settings on *discover*

We installed a Python distribution. To use it, you need to load the modules:

```
module load other/comp/gcc-4.5-sp1
module load lib/mkl-10.1.2.024
module load other/SIVO-PyD/spd_1.7.0_gcc-4.5-sp1
```

SIVO-PyD

- Collection of Python packages for scientific computing and visualization
- All the packages are accessible within the Python framework
- Self-contained distribution.

https://modelingguru.nasa.gov/docs/DOC-2109

Settings on your Local Mac

Go to the following Modeling Guru page:

https://modelingguru.nasa.gov/docs/DOC-1847

and follow the instructions to install Python, Numpy, SciPy, Matplotlib and Basemap.

What Will be Covered in the Four Sessions

1. Python
2. Numpy
3. SciPy
4. netCDF4
5. Matplotlib and Basemap
6. EOFs

What Will be Covered Today

1. **Simple Program**
2. **Print Statement**
3. **Python Expression**
4. **Loops**
5. **Defining a Function**
6. **Basic I/O**
7. **iPython**
8. The os Module
9. Creating your own Module
10. **List**
11. Tuples
12. Dictionary
13. Classes in Python

What is Python?

Python is an elegant and robust programming language that combines the power and flexibility of traditional compiled languages with the ease-of-use of simpler scripting and interpreted languages.

What is Python?

- High level
- Interpreted
- Scalable
- Extensible
- Portable
- Easy to learn, read and maintain
- Robust
- Object oriented
- Versatile

Why Python?

- Free and Open source
- Built-in run-time checks
- Nested, heterogeneous data structures
- OO programming
- Support for efficient numerical computing
- Good memory management
- Can be integrated with C, C++, Fortran and Java
- Easier to create stand-alone applications on any platform

Scientific Hello World

- Provide a number to the script
- Print 'Hello World' and the *sine* value of the number

To run the script, type:

```
./helloWorld.py 3.14
```

Purpose of the Script

- Read a command line argument
- Call a math (sine) function
- Work with variables
- Print text and numbers

The Code

```
1  #!/usr/bin/env python
2
3  import sys
4  import math
5
6  r = float(sys.argv[1])
7  s = math.sin(r)
8  print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

Header

- **Explicit path to the interpreter:**

```
#!/usr/bin/python
#!/usr/local/other/Python-2.5.4/bin/python
```

- **Using env to find the first Python interpreter in the path:**

```
#!/usr/bin/env python
```

Importing Python Modules

The standard way of loading a module is:

```
import scipy
```

We can also use:

```
from scipy import *
```

We may choose to load a sub-module of the main one:

```
import scipy.optimize
from scipy.optimize import *
```

We can choose to retrieve a specific function of a module:

```
from scipy.optimize import fsolve
```

You can even rename a module:

```
import scipy as sp
```

Alternative Print Statements

- **String concatenation:**

```
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

- **C printf-like statement:**

```
print "Hello, World! sin(%g)=%g" % (r,s)
```

- **Variable interpolation:**

```
print "Hello, World! sin(%(r)g)=%(s)g" % vars()
```

Printf Format Strings

%d : integer

%5d : integer in a field of width 5 chars

%-5d : integer in a field of width 5 chars, but adjusted to the left

%05d : integer in a field of width 5 chars, padded with zeroes from the left

%g : float variable in

%e : float variable in scientific notation

%11.3e : float variable in scientific notation, with 3 decimals, field of width 11 chars

%5.1f : float variable in fixed decimal notation, with one decimal, field of width 5 chars

%.3f : float variable in fixed decimal form, with three decimals, field of min. width

%s : string

%-20s : string in a field of width 20 chars, and adjusted to the left

Python Types

- Numbers: float, complex, int (+ bool)
- Sequences: list, tuple, str, NumPy arrays
- Mappings: dict (dictionary/hash)
- Instances: user-defined class
- Callables: functions, callable instances

Numerical Expressions

- **Python distinguishes between strings and numbers:**

```
b =  1.2            # b is a number
b = '1.2'           # b is a string
a = 0.5 * b         # illegal: b is NOT converted to float
a = 0.5 * float(b) # this works
```

- **All Python objects are compared with:**

```
==   !=  <  >  <=  >=
```

Boolean Expressions

- bool is True or False
- Can mix bool with int 0 (false) or 1 (true)
- Boolean tests:

```
a = '';  a = [];  a = ();  a = ;  # empty structures
a = 0;   a = 0.0
if a:                             # false
if not a:                         # true
```

other values of a: if a is true

Strings

- **Single- and double-quoted strings work in the same way:**

```
s1 = "some string with a number %g" % r
s2 = 'some string with a number %g' % r       # = s1
```

- **Triple-quoted strings can be multi line with embedded newlines:**

```
text = """
large portions of a text can be conveniently
placed inside triple-quoted strings
(newlines are preserved)"""
```

- **Raw strings, where backslash is backslash:**

```
s3 = r"\(\s+\.\d+\)"
# with ordinary string (must quote backslash):
s3 = '\\(\\s+\\.\\d+\\)'
```

Variables and Data Types

Type	Range	To Define	To Covert
float	numbers	x=1.0	z=float(x)
integer	numbers	x=1	z=int(x)
complex	complex numbers	x=1+3j	z=complex(a,b)
string	text string	x='test'	z=str(x)
boolean	True or False	x=True	z=bool(x)

If Statements

```
if <conditions>:
    <stattements>
elif <conditions>:
    <statements>
else:
    <statements>
```

```
1  x = 10
2  if x > 0:
3      print 1
4  elif x == 0:
5      print 0
6  else:
7      print    1
```

For Loops

For loops iterate over a sequence of objects:

```
for <loop_var> in <sequence>:
    <statements>
```

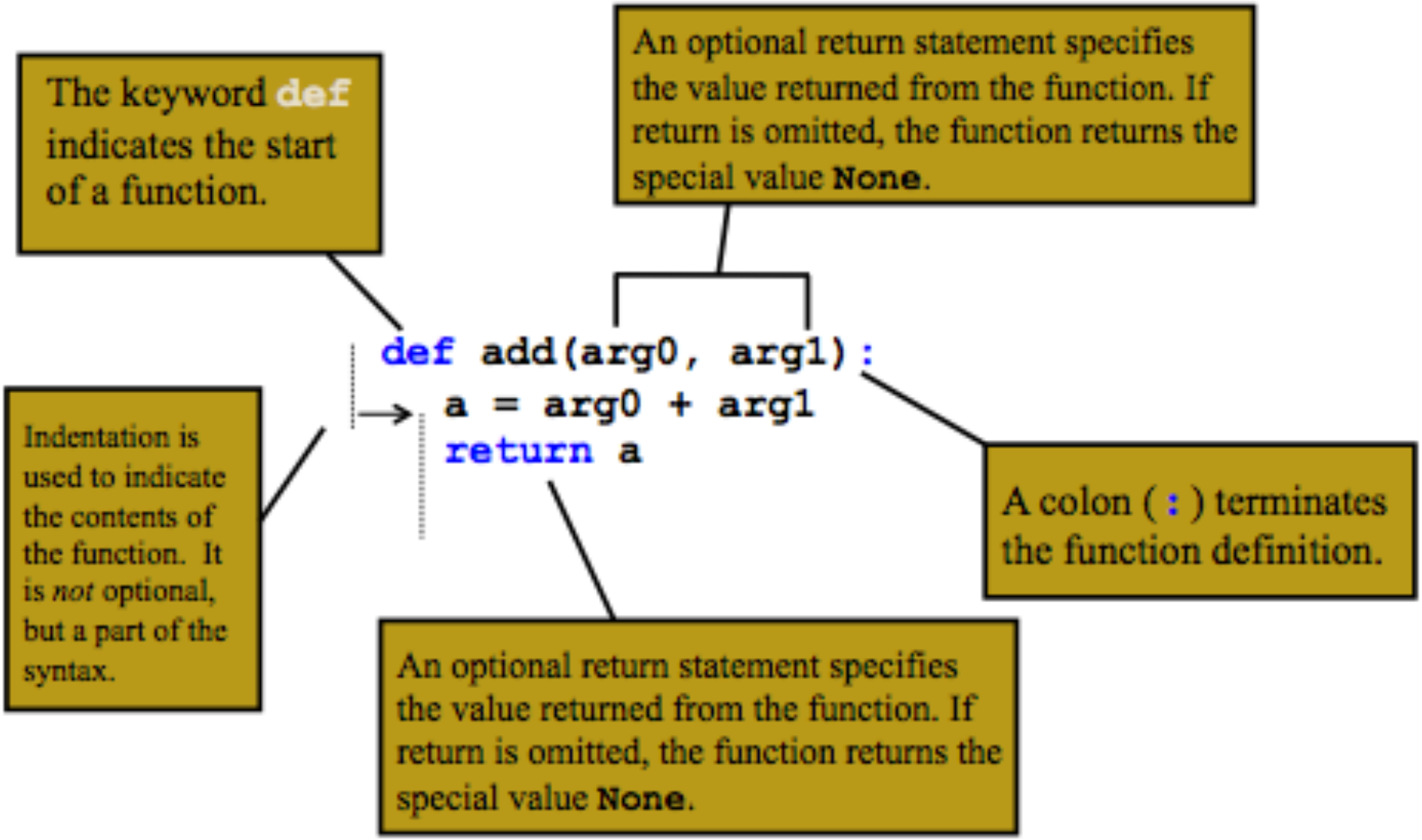
```
1  for i in range(5):
2      print i,
3
4  for i in "abcde":
5      print i,
6
7  l=["dogs","cats","bears"]
8  accum = " "
9  for item in l:
10     accum = accum + item
11     accum = accum + " "
```

While Loop

```
while <condition>:
      <statements>
```

```
1  lst = range(3)
2  while lst:
3      print lst
4      lst = lst[1:]
5
6  i = 0
7  while 1:
8      if i < 3:
9          print i,
10     else:
11         break
12     i = i + 1
```

Functions


The keyword def indicates the start of a function.
An optional return statement specifies the value returned from the function. If return is omitted, the function returns the special value None.
def add(arg0, arg1):
a = arg0 + arg1
return a
Indentation is used to indicate the contents of the function. It is *not* optional, but a part of the syntax.
A colon (:) terminates the function definition.
An optional return statement specifies the value returned from the function. If return is omitted, the function returns the special value None.

Reading/Writing Data Files

Task:

- Read (x,y) data from a two-column file
- Transform y values to f(y)
- Write (x,f(y)) to a new file

What to learn:

- How to open, read, write and close file
- How to write and call a function
- How to work with arrays (lists)

Reading Input/Output Filenames

- Usage:

```
./readInputOutputFiles1.py inFile outFile
```

- Read the two command-line arguments: input and output filenames

```
infilename  = sys.argv[1]
outfilename = sys.argv[2]
```

- Command-line arguments are in sys.argv[1:]
- sys.argv[0] is the name of the script

Exception Handling

- What if the user fails to provide two command-line arguments?
- Python aborts execution with an informative error message
- Manual handling of errors:

```
1  try:
2      infilename  = sys.argv[1]
3      outfilename = sys.argv[2]
4  except:
5      # try block failed, we miss two command-line argum
6      print 'Usage:', sys.argv[0], 'inFile outFile'
7      sys.exit(1)
```

Open File and Read Line by Line

- Open files:

```
ifile = open( infilename, 'r')  # r for reading
ofile = open(outfilename, 'w')  # w for writing
afile = open(appfilename, 'a')  # a for appending
```

- Read line by line

```
for line in ifile:
    # process line
```

- Observe: blocks are indented; no braces!

Defining a Function

```
1   import math
2   def myfunc(y):
3       if y >= 0.0:
4           return y**5*math.exp(-y)
5       else:
6           return 0.0
7
8   # alternative way of calling module functions
9   # (gives more math-like syntax in this example):
10
11  from math import *
12  def anotherfunc(y):
13      if y >= 0.0:
14          return y**5*exp(-y)
15      else:
16          return 0.0
```

Data Transformation Loop

- Input file format: two columns with numbers

```
0.1   1.4397
0.2   4.325
0.5   9.0
```

- Read (x,y), transform y, write (x,f(y)):

```
1    for line in ifile:
2        pair = line.split()
3        x = float(pair[0])
4        y = float(pair[1])
5        fy = myfunc(y)  # transform y value
6        ofile.write('%g  %12.5e\n' % (x,fy))
```

Alternative File Reading

This construction is more flexible and traditional in Python:

```
1    while 1:
2        line = ifile.readline()  # read a line
3        if not line: break
4        # process line
```

i.e., an 'infinite' loop with the termination criterion inside the loop

Loading Data into Lists

- Read input file into list of lines:

```
lines = ifile.readlines()
```

- Now the 1st line is lines[0], the 2nd is lines[1], etc.
- Store x and y data in lists:

```
# go through each line, split line into x and y colum
x = []
y = []
for line in lines:
    xval, yval = line.split()
    x.append(float(xval))
    y.append(float(yval))
```

Loop over List Entries

Loop over (x,y) values:

```
1    ofile = open(outfilename, 'w') # open for writing
2    for i in range(len(x)):
3        fy = myfunc(y[i])  # transform y value
4        ofile.write('%g  %12.5e\n' % (x[i], fy))
5    ofile.close()
```

Computing with Arrays

- x and y in `readInputOutputFiles2.py` are lists
- We can compute with lists element by element (as shown)
- However: using Numerical Python (NumPy) arrays instead of lists is much more efficient and convenient
- Numerical Python is an extension of Python: a new fixed-size array type and lots of functions operating on such arrays

Interactive Computing with ipython

- Allows to run commands interactively
- Prompts you to executes any valid Python statement
- Executes Python scripts: run myFile.py args
- Points to any error
- Provides help functionality: help numpy.random
- Up- and down-arrows: go through command history
- The underscore variable holds the last output

Ipython - TAB Completion

IPython supports TAB completion: write a part of a command or name (variable, function, module), hit the TAB key, and IPython will complete the word or show different alternatives

```
In [1]: import math
In [2]: math.<TABKEY>
math.__class__      math.__str__      math.frexp
math.__delattr__    math.acos         math.hypot
math.__dict__       math.asin         math.ldexp...
```

or

```
In [2]: my_variable_with_a_very_long_name = True
In [3]: my<TABKEY>
In [3]: my_variable_with_a_very_long_name
```

You can increase your typing speed with TAB completion!

The os Module

- Python has a rich cross-platform operating system (OS) interface
- Skip Unix- or DOS-specific commands; do all OS operations in Python!
- The os module provides dozens of functions for interacting with the operating system

```
import os
dir(os)
help(os)
```

Some os Functions

```
curDir = os.getcwd()                    # Return the current working di
os.chdir('targetDir')                   # Change directory
os.remove('file')                       # Remove file
os.rename('oldName', 'newName') # Rename file
os.listdir('myDir')                     # Provide a list of files in myl
os.removedirs('myDir')                  # Remove all empty directories a
os.system('command')                    # Execute the command
```

Creating a Subdirectory

```
1       dir = case                          # subdirectory name
2       import os, shutil
3       if os.path.isdir(dir):              # does dir exist?
4           shutil.rmtree(dir)              # yes, remove old files
5       os.mkdir(dir)                       # make dir directory
6       os.chdir(dir)                       # move to dir
```

A Simple Module

```
1  #!/usr/bin/env python
2  # FileName: mymodule.py
3
4  def sayhi(name):
5      print 'Hi from %s: this is mymodule speaking.' %
6
7  version = '0.1'
```

- Remember that the module should be placed in the same directory as the program that we import it in, or
- The module should be in one of the directories listed in *sys.path*.

Use the Module

```
1  #!/usr/bin/env python
2  # Filename: mymodule_demo.py
3
4  import mymodule
5
6  mymodule.sayhi(  Jules  )
7  print 'Version', mymodule.version
```

If you run the above script:

```
./mymodule_demo.py
```

You will get

```
Hi from Jules: this is mymodule speaking.
```

Executing Modules as Scripts-1

- We want to execute the code in the module as it was imported
- We need to add the following at the end of the module:

```
1    if __name__ == "__main__":
2        # section of the module to be executed
```

Executing Modules as Scripts-2

```
#!/usr/bin/env python
# FileName: mymodule.py

def sayhi(name):
    print "Hi from %s: this is mymodule speaking." %
version = '1.0'

if __name__ == "__main__":
    import sys
    try:
        sayhi(str(sys.argv[1]))
    except:
        print 'Usage: %s string' % sys.argv[0]
        sys.exit(0)
    print 'Version', version
```

Setting List Elements

- Initializing a list:

```
arglist = [myarg1, 'displacement', "tmp.ps"]
```

- Or with indices (if there are already two list elements):

```
arglist[0] = myarg1
arglist[1] = 'displacement'
```

- Create list of specified length:

```
n = 100
mylist = [0.0]*n
```

- Adding list elements:

```
arglist = []   # start with empty list
arglist.append(myarg1)
arglist.append('displacement')
```

Getting List Elements

- Extract elements from a list:

```
filename, plottitle, psfile  = arglist
(filename, plottitle, psfile) = arglist
[filename, plottitle, psfile] = arglist
```

- Or with indices:

```
filename = arglist[0]
plottitle = arglist[1]
```

Traversing Lists

- For each item in a list:

```
for entry in arglist:
    print 'entry is', entry
```

- For-loop-like traversal:

```
start = 0
stop = len(arglist)
step = 1
for index in range(start, stop, step):
    print 'arglist[%d]=%s' % (index,arglist[index])
```

- Visiting items in reverse order:

```
mylist.reverse()            # reverse order
for item in mylist:
    # do something...
```

List Comprehensions

- Compact syntax for manipulating all elements of a list:

```
y = [ float(yi) for yi in line.split() ]  # call function f
x = [ a+i*h for i in range(n+1) ]         # execute express
```

(called list comprehension)

- Written out:

```
y = []
for yi in line.split():
    y.append(float(yi))
etc.
```

Map Function

- map is an alternative to list comprehension:

```
y = map(float, line.split())
x = map(lambda i: a+i*h, range(n+1))
```

- map is faster than list comprehension but not as easy to read

Typical List Operations

```
d = []              # declare empty list
d.append(1.2)       # add a number 1.2
d.append('a')      # add a text
d[0] = 1.3          # change an item
del d[1]            # delete an item
len(d)              # length of list
d.count(x)          # count the number of times x occurs
d.index(x)          # return the index of the first occurrence of x
d.remove(x)         # delete the first occurrence of x
d.reverse           # reverse the order of elements in the list
```

Nested List

- Lists can be nested and heterogeneous
- List of string, number, list and dictionary:

```
>>> mylist = ['t2.ps', 1.45, ['t2.gif', 't2.png'],\
     { 'factor' : 1.0, 'c' : 0.9} ]
>>> mylist[3]
{'c': 0.90000000000000002, 'factor': 1.0}
>>> mylist[3]['factor']
1.0
>>> print mylist
['t2.ps', 1.45, ['t2.gif', 't2.png'],
{'c': 0.90000000000000002, 'factor': 1.0}]
```

- Note: print prints all basic Python data structures in a nice format

Sorting a List

- In-place sort:

```
mylist.sort()

# modifies mylist!

>>> print mylist
[1.4, 8.2, 77, 10]
>>> mylist.sort()
>>> print mylist
[1.4, 8.2, 10, 77]
```

- Strings and numbers are sorted as expected

Defining the Comparison Criterion

```
# ignore case when sorting:
def ignorecase_sort(s1, s2):
    s1 = s1.lower()
    s2 = s2.lower()
    if   s1 <  s2: return -1
    elif s1 == s2: return  0
    else:          return  1

# or a quicker variant, using Python's built-in cmp function:

def ignorecase_sort(s1, s2):
    s1 = s1.lower();  s2 = s2.lower()
    return cmp(s1,s2)

# usage:
mywords.sort(ignorecase_sort)
```

Indexing

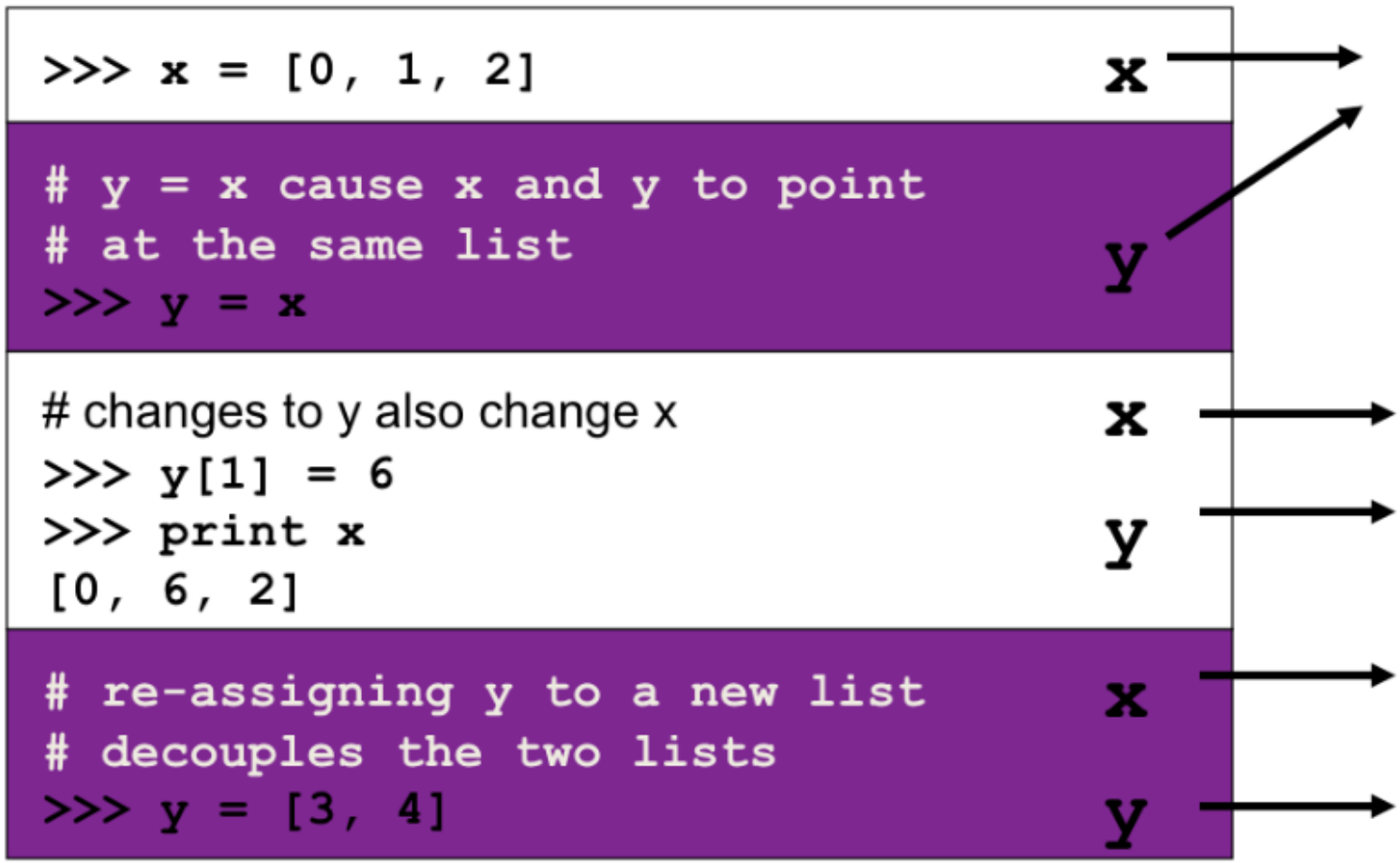
```
# list
# indices: 0 1 2 3 4
>>> l = [10,11,12,13,14]
>>> l[0]
10

# negative indices count backward from
# the end of the list
# indices: -5 -4 -3 -2 -1
>>> l = [10,11,12,13,14]
>>> l[-1]
14
>>> l[-2]
13
```

Slicing

var[lower:upper]
Slices extract a portion of a sequence by specifying a lower and upper bound. The extracted elements start at lower and go up to, but do not include, the upper element. Mathematically the range is [lower,upper).

```
>>> l = [10,11,12,13,14]
>>> l[1:3]
[11,12]
>>> l[1,-2]
[11,12]
>>> l[-4:3]
[11,12]
>>> l[:3]           # grab the first three elements
[10,11,12]
>>> l[-2:]          # grab the last two elements
[13,14]
```

Assignment Creates Object Reference



What are Tuples

- Tuples are a sequence of objects just like lists.
- Unlike lists, tuples are immutable objects.
- A good rule of thumb is to use lists whenever you need a generic sequence.

Examples of Tuples

- Tuple = constant list; items cannot be modified

```
>>> s1=[1.2, 1.3, 1.4]          # list
>>> s2=(1.2, 1.3, 1.4)          # tuple
>>> s2=1.2, 1.3, 1.4            # may skip parenthesis
>>> s1[1]=0                     # ok
>>> s2[1]=0                     # illegal
Traceback (innermost last):
  File "<pyshell#17>", line 1, in ?
    s2[1]=0
TypeError: object doesn't support item assignment

>>> s2.sort()
AttributeError: 'tuple' object has no attribute 'sort'
```

- You cannot append to tuples, but you can add two tuples to form a new tuple

What is a Dictionary?

- Dictionary = array with a text as index
- Also called hash or associative array in other languages
- Can store 'anything:

```
prm['damping'] = 0.2                    # number
def x3(x):
    return x*x*x
prm['stiffness'] = x3                   # function object

prm['model1'] = [1.2, 1.5, 0.1]         # list object
```

- The text index is called key

Dictionary Operations

- Dictionary = array with text indices (keys, even user-defined objects can be indices!)
- Also called hash or associative array
- Common operations:

```
d['mass']           # extract item corresp. to key 'mass
d.keys()            # return copy of list of keys
d.get('mass',1.0) # return 1.0 if 'mass' is not a key
d.has_key('mass') # does d have a key 'mass'?
d.values()          # return a list of all values in the dictionary
d.items()           # return list of (key,value) tuples
d.copy()            # create a copy of the dictionary
d.clear()           # remove all key/value pair from the dictionary
del d['mass']       # delete an item
len(d)              # the number of items
```

Initializing Dictionaries

- Multiples items:

```
d = { 'key1' : value1, 'key2' : value2 }
# or
d = dict(key1=value1, key2=value2)
```

- Item by item (indexing):

```
d['key1'] = anothervalue1
d['key2'] = anothervalue2
d['key3'] = value2
```

Example of Dictionary

```
1   # create an empty dictionary using curly brackets
2   >>> record = {}
3   >>> record['first'] = 'Jmes'
4   >>> record['last'] = 'Maxwell'
5   >>> record['born'] = 1831
6   >>> print record
7   {'first': 'Jmes', 'born': 1831, 'last': 'Maxwell'}
8   # create another dictionary with initial entries
9   >>> new_record = {'first': 'James', 'middle':'Clerk'}
10  # now update the first dictionary with values from th
11  >>>record.update(new_record)
12  >>> print record
13  {'first': 'James', 'middle': 'Clerk', 'last':'Maxwell
```

Another Example of Dictionary

- Problem: store MPEG filenames corresponding to a parameter with values 1, 0.1, 0.001, 0.00001

```
movies[1]          = 'heatsim1.mpeg'
movies[0.1]        = 'heatsim2.mpeg'
movies[0.001]      = 'heatsim5.mpeg'
movies[0.00001]    = 'heatsim8.mpeg'
```

- Store compiler data:

```
g77 = {
'name'             : 'g77',
'description'      : 'GNU f77 compiler, v2.95.4',
'compile_flags'    : ' -pg',
'link_flags'       : ' -pg',
'libs'             : '-lf2c',
'opt'              : '-O3 -ffast-math -funroll-loops'
     }
```

Sample Dictionary

Check the file:

`sampleDictionary.py`

and run it.

Introduction of Classes in Python

- Similar class concept as in Java and C++
- All functions are virtual
- No private/protected variables (the effect can be "simulated")
- Single and multiple inheritance
- Everything in Python is a class and works with classes
- Class programming is easier and faster than in C++ and Java (?)

The Basics of Python Classes

- Declare a base class MyBase:

```
class MyBase:
    def __init__(self,i,j):  # constructor
        self.i = i; self.j = j

    def write(self):         # member function
        print 'MyBase: i=',self.i,'j=',self.j
```

- self is a reference to this object
- Data members are prefixed by self:
  self.i, self.j
- All functions take self as first argument in the declaration, but not in the call

```
obj1 = MyBase(6,9)
obj1.write()
```

Implementing a Subclass

- Class MySub is a subclass of MyBase:

```
class MySub(MyBase):
    def __init__(self,i,j,k):  # constructor
        MyBase.__init__(self,i,j)
        self.k = k;
    def write(self):
        print 'MySub: i=',self.i,'j=',self.j,'k=',self.k
```

- Example:

```
# this function works with any object that has a write func:
def write(v): v.write()

# make a MySub instance
i = MySub(7,8,9)
write(i)   # will call MySub's write
```

References I


Hans Petter Langtangen.
A Primer on Scientific Programming with Python.
Springer, 2009.


Johnny Wei-Bing Lin.
AHands-On Introduction to Using Python in the Atmospheric and Oceanic Sciences.
http://www.johnny-lin.com/pyintro, 2012.


Drew McCormack.
Scientific Scripting with Python.
2009.


Sandro Tosi.
Matplotlib for Python Developers.
2009.