Sandia
National
Laboratories

Exceptional
service
in the
national
interest

2. A Python Tutorial

PYOMO

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

CCR Center for Computing Research

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. SAND NO. 2011-XXXXP

Why Python?

- Interpreted language
- Intuitive syntax
- Dynamic typing
- Lots of built-in libraries and third-party extensions
- Shallow learning curve
- Integration with C/Java
- Object-oriented
- Simple, but extremely powerful

Python Implementations

- Cpython
  - C Python interpreter
  - https://www.python.org/downloads/
  - SciPy Stack
    - http://www.scipy.org/install.html
    - Anaconda: Linux/MacOS/MS Windows
- PyPy
  - A Python interpreter written in Python
  - http://pypy.org/
- Jython
  - Java Python interpreter
  - http://www.jython.org/
- IronPython
  - .NET Python interpreter
  - http://ironpython.net/

Full Pyomo Support

Beta Pyomo Support

Pyomo Not Supported (yet)

Python Versions: 2.x *vs* 3.x

- Python 3.0 was released in 2008
  - Included significant backward incompatibilities

- Adoption of Python 3.x has been slow
  - Major Linux distributions are still including Python 2.x
  - Major Python packages have slowly transitioned
  - Some commercial packages still only have Python 2.x interfaces

- Status
  - Python 2.7.11
    - Very stable; patches have included package updates to support Python 3.x compatibility
  - Python 3.5.1
    - Very stable

We try to stick to “universal” syntax that will work in both 2.x and 3.x

Overview

- interactive "shell"
- basic types: numbers, strings
- container types: lists, dictionaries, tuples
- variables
- control structures
- functions & procedures
- classes & instances
- modules
- exceptions
- files & standard library

Interactive Shell

- Great for learning the language
- Great for experimenting with the library
- Great for testing your own modules
- Two variations:
  - IDLE (GUI)
  - python (command line)
- Type statements or expressions at prompt:

```
>>> print( "Hello, world" )
Hello, world
>>> x = 12**2
>>> x/2
72
>>> # this is a comment
```

Python Program

- To write a program, put commands in a file

```
# hello.py
print( "Hello, world" )
x = 12**2
print( x )
```

- Execute on the command line

```
C:\Users\me> python hello.py
Hello, world
144
```

Python Variables

- No need to declare
- Need to assign (initialize)
  - use of uninitialized variable raises exception
- Not typed

```
greeting = 34.2
if friendly:
    greeting = "hello world"
else:
    greeting = 12**2
print( greeting )
```

- ***Everything*** is a "variable":
  - Even functions, classes, modules

Control Structures

```
if condition:
    statements
[elif condition:
    statements] ...
else:
    statements
```

```
while condition:
    statements

for var in sequence:
    statements

break
continue
```

Note: *Spacing matters!*

Control structure scope dictated by indentation

Grouping Indentation

In Python:

```python
for i in range(20):
    if i % 3 == 0:
        print(i)
        if i % 5 == 0:
            print("Bingo!“)
    print("---")
```

In C:

```c
for (i = 0; i < 20; i++)
{
   if (i % 3 == 0) {
      printf("%d\n", i);
      if (i % 5 == 0) {
         printf("Bingo!\n");
      }
   }
   printf("---\n");
}
```

Numbers

- The usual suspects
    - `12,   3.14,   0xFF,   0377,   (-1+2)*3/4**5,   abs(x),   0<x<=5`
- C-style shifting & masking
    - `1<<16,   x&0xff,   x|1,   ~x,   x^y`
- Integer division truncates
    - Python 2.x
        - `1/2` → `0,   1./2.` → `0.5,   float(1)/2` → `0.5`
        - `from __future__ import division`
            - » `1/2` → `0.5`
    - Python 3.x
        - `1/2` → `0.5`
- Long (arbitrary precision), complex
    - `2L**100` → `1267650600228229401496703205376L`
        - In Python 2.2 and beyond, 2**100 does the same thing
    - `1j**2` → `(-1+0j)`

Strings

"hello"+"world”	"helloworld”	*# concatenation*
"hello"*3	"hellohellohello"	*# repetition*
"hello"[0]	"h"	*# indexing*
"hello"[-1]	"o"	*# (from end)*
"hello"[1:4]	"ell"	*# slicing*
len("hello")	5	*# size*
"hello" < "jello”	True	*# comparison*
"e" in "hello”	True	*# search*

- "escapes: \n etc, \033 etc, \if etc"
- 'single quotes' """triple quotes""" r"raw strings"

Lists

- Flexible arrays, *not* linked lists
  - `a = [99, "bottles of beer", ["on", "the", "wall"]]`

- Same operators as for strings
  - `a+b`, `a*3`, `a[0]`, `a[-1]`, `a[1:]`, `len(a)`

- Item and slice assignment
  - `a[0] = 98`
  - `a[1:2] = ["bottles", "of", "beer"]`

    ```
    # -> [98, "bottles", "of", "beer", ["on", "the", "wall"]]
    ```
  - `del a[-1]`

    ```
    # -> [98, "bottles", "of", "beer"]
    ```

List Operations

```
>>> a = range(5)           # [0,1,2,3,4]
>>> a.append(5)            # [0,1,2,3,4,5]
>>> a.pop()                # [0,1,2,3,4]
5
>>> a.insert(0, 42)        # [42,0,1,2,3,4]
>>> a.pop(0)               # [0,1,2,3,4]
42
>>> a.reverse()            # [4,3,2,1,0]
>>> a.sort()               # [0,1,2,3,4]
```

Dictionaries

- Hash tables, "associative arrays"
  - `d = {"duck": "eend", "water": "water"}`

- Lookup:
  - `d["duck"]     # -> "eend"`
  - `d["back"]     # raises KeyError exception`

- Delete, insert, overwrite:
  - `del d["water"]        # {"duck": "eend", "back": "rug"}`
  - `d["back"] = "rug"     # {"duck": "eend", "back": "rug"}`
  - `d["duck"] = "duik"    # {"duck": "duik", "back": "rug"}`

Dictionary Operations

- Keys, values, items:

```
d.keys()     -> ["duck", "back"]
d.values()   -> ["duik", "rug"]
d.items()    -> [("duck","duik"), ("back","rug")]
```

Note: These actually return generators, not lists.

- Presence check:

```
d.has_key("duck")   # -> 1; d.has_key("spam") -> 0
```

- Values of any type; keys almost any

```
{ "name": "Guido",
  "age": 43,
  ("hello","world"): 1,
  42: "yes",
  "flag": ["red","white","blue"] }
```

Dictionary Details

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - keys are hashed (to ensure fast lookup)
  - lists or dictionaries cannot be used as keys
    - these objects can be changed "in place"
  - no restrictions on values

- Keys will be listed in **arbitrary order**
  - key hash values are in an arbitrary order
  - that numeric keys are returned sorted is an artifact of the implementation and *is not guaranteed*

Tuples

```
key = (lastname, firstname)
point = x, y, z             # parentheses optional
x, y, z = point             # unpack
lastname = key[0]           # index tuple values
singleton = (1,)            # trailing comma!!!
                            #   (1) → integer!
empty = ()                  # parentheses!
```

- Tuples vs. lists
  - tuples immutable
  - lists mutable

Reference Semantics

- Assignment manipulates references
  - `x = y` **does not make a copy** of y
  - `x = y` makes x **reference** the object y references
- Reference values can be modified!

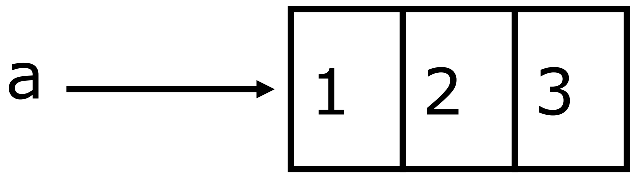
```
>>> a = [1, 2, 3]
>>> b = a
>>> a.append(4)
>>> print(b)
[1, 2, 3, 4]
```

- Copied objects are distinct

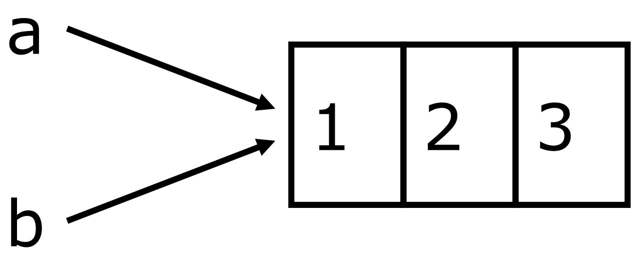
```
>>> import copy
>>> c = copy.copy(a)
>>> a.pop()
>>> print(c)
[1, 2, 3, 4]
```

Changing a Shared List

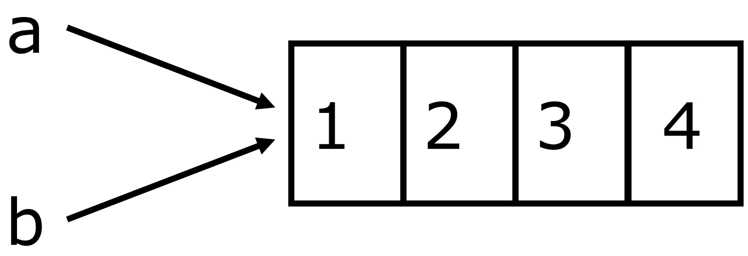
```
a = [1, 2, 3]
```



```
b = a
```



```
a.append(4)
```

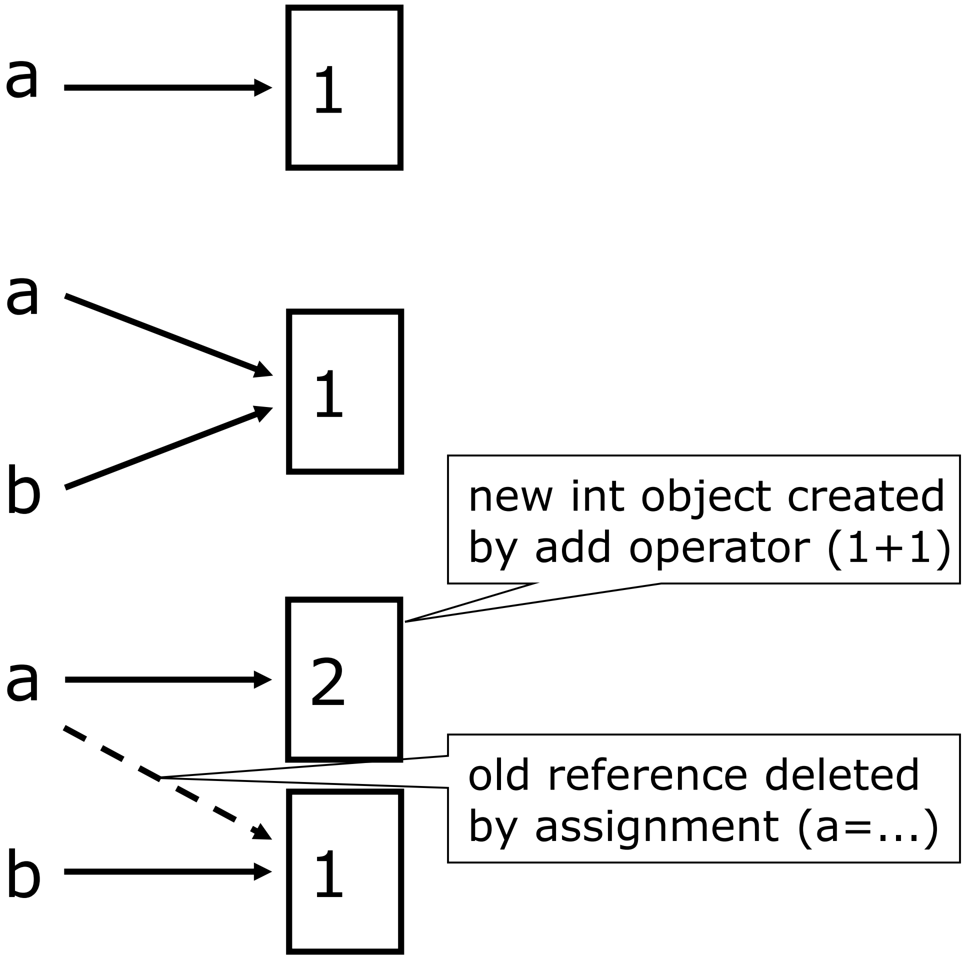


Changing an Integer

```
a = 1
```

```
b = a
```

```
a = a+1
```



new int object created by add operator (1+1)

old reference deleted by assignment (a=...)

Functions / Procedures

```
def name(arg1, arg2, ...):
    """documentation"""   # optional doc string
    statements

    return expression     # from function
    return                # from procedure (returns None)
```

Example

```
def gcd(a, b):
    """greatest common divisor"""
    while a != 0:
        a, b = b%a, a   # parallel assignment
    return b

>>> gcd.__doc__
'greatest common divisor'

>>> gcd(12, 20)
4
```

Classes

```
class name(object):
    """documentation"""
    statements
```

Most, *statements* are method definitions:

```
    def name(self, arg1, arg2, ...):
        ...
```

May also be *class variable* assignments

Example

```
class Stack(object):
    """A well-known data structure..."""
    def __init__(self):             # constructor
        self.items = []
    def push(self, x):
        self.items.append(x)        # the sky is the limit
    def pop(self):
        x = self.items[-1]          # what if it’s empty?
        del self.items[-1]
        return x
    def empty(self):
        return len(self.items) == 0       # Boolean result
```

Example (cont'd)

- To create an instance, simply call the class object:

```
x = Stack()                # no 'new' operator!
```

- To use methods of the instance, call using dot notation:

```
x.empty()                  # -> 1
x.push(1)                  # [1]
x.empty()                  # -> 0
x.push("hello")            # [1, "hello"]
x.pop()                    # -> "hello"   # [1]
```

- To inspect instance variables, use dot notation:

```
x.items                    # -> [1]
```

Class/Instance Variables

```
class Connection(object):
    verbose = 0                 # class variable

    def __init__(self, host):
        self.host = host        # instance variable

    def debug(self, v):
        self.verbose = v        # make instance variable!

    def connect(self):
        if self.verbose:        # class or instance variable?
            print("connecting to %s" % (self.host,))
```

Instance Variable Rules

- On use via instance (`self.x`), search order:
  - (1) instance, (2) class, (3) base classes
  - this also works for method lookup
- On assignment via instance (`self.x = ...`):
  - always makes an instance variable
- Class variables "default" for instance variables
- But...!
  - mutable *class* variable: one copy *shared* by all
  - mutable *instance* variable: each instance its own

Modules

- Collection of stuff in *foo*.py file
  - functions, classes, variables

- Importing modules:

```
import re
print( re.match("[a-z]+", s) )
from re import match
print( match("[a-z]+", s) )
```

- Import with rename:

```
import re as regex
from re import match as m
```

Catching Exceptions

```
def foo(x):
    return 1/x

def bar(x):
    try:
        print( foo(x) )
    except ZeroDivisionError as message:
        print("Can’t divide by zero: %s" % message)

bar(0)
```

Try-Finally: Cleanup

```
f = open(file)
try:
    process_file(f)
finally:
    f.close()               # always executed
print("OK")                 # executed on success only
```

Raising Exceptions

```
raise IndexError

raise IndexError("k out of range")

raise IndexError, "k out of range”
           # this only works in Python 2.x!

try:
    something
except:                    # catch everything
    print( "Oops" )
    raise                  # reraise
```

More on Exceptions

- User-defined exceptions
  - subclass `Exception` or any other standard exception

- Note: in older versions of Python exceptions can be strings

- Last caught exception info:
  - `sys.exc_info() == (exc_type, exc_value, exc_traceback)`

- Printing exceptions: traceback module

Major Python Packages

- SciPy
  - Scientific Python for mathematics and engineering
  - http://www.scipy.org
- Numpy
  - Numeric array package
  - http://www.numpy.org/
- Matplotlib
  - 2D plotting library
  - http://matplotlib.org/
- Pandas
  - Data structures and analysis
  - http://pandas.pydata.org/
- Ipython
  - Interactive Python shell
  - http://ipython.org/

Resources

- Software Carpentry
  - http://software-carpentry.org/
- Python webpage
  - http://www.python.org
- Books
  - Python Essential Reference (4th Edition), David Beazley, 2009
  - Python in a Nutshell, Alex Martelli, 2003
  - Python Pocket Reference, 4th Edition, Mark Lutz, 2009
  - …

Acknowledgements

- William Hart
- Ted Ralphs
- John Siirola
- Dave Woodruff
- Guido van Rossum