Computational Integer Programming

Lecture 4: Python

Dr. Ted Ralphs

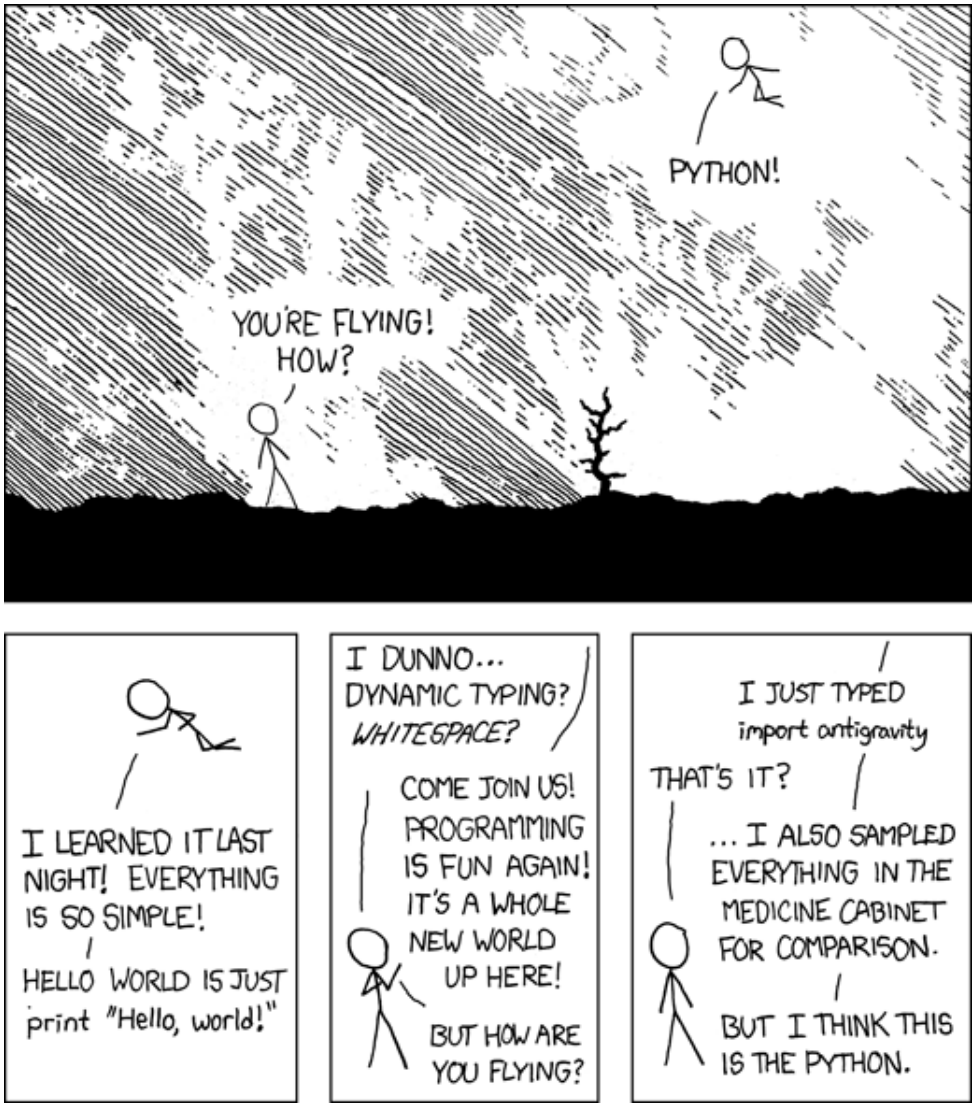
LEHIGH
UNIVERSITY

COR@L
COMPUTATIONAL OPTIMIZATION
RESEARCH AT LEHIGH

Why Python?

- Pros
  - As with many high-level languages, development in Python is quick and painless (relative to C++!).
  - Python is popular in many disciplines and there is a dizzying array of packages available.
  - Python's syntax is very clean and naturally adaptable to expressing mathematical programming models.
  - Python has the primary data structures necessary to build and manipulate models built in.
  - There has been a strong movement toward the adoption of Python as the high-level language of choice for (discrete) optimizers.
  - Sage is quickly emerging as a very capable open-source alternative to Matlab.
- Cons
  - Python's one major downside is that it can be very slow.
  - Solution is to use Python as a front-end to call lower-level tools.

Drinking the Python Kool-Aid



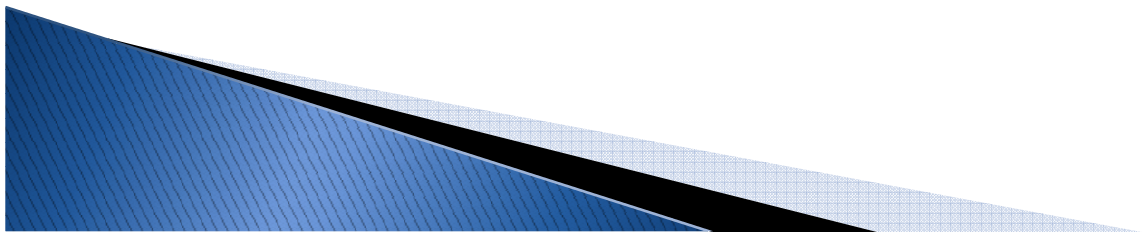
Introduction to Python

Adapted from a Tuturial by Guido van Rossum
Director of PythonLabs at Zope Corporation

Presented at
LinuxWorld - New York City - January 2002

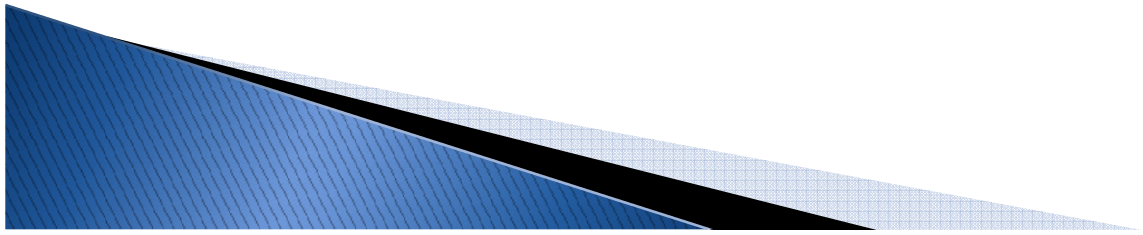
Why Python?

- Interpreted language
- Intuitive syntax
- Dynamic typing
- **Loads** of built-in libraries and available extensions
- Shallow learning curve
- Easy to call C/C++ for efficiency
- Object-oriented
- Simple, but extremely powerful



Tutorial Outline

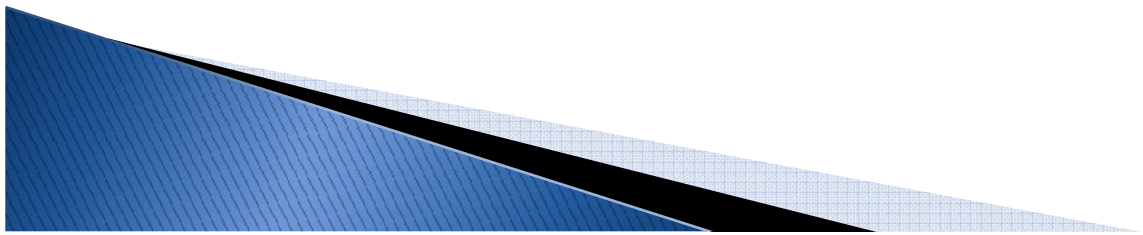
- interactive "shell"
- basic types: numbers, strings
- container types: lists, dictionaries, tuples
- variables
- control structures
- functions & procedures
- classes & instances
- modules
- exceptions
- files & standard library



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
```



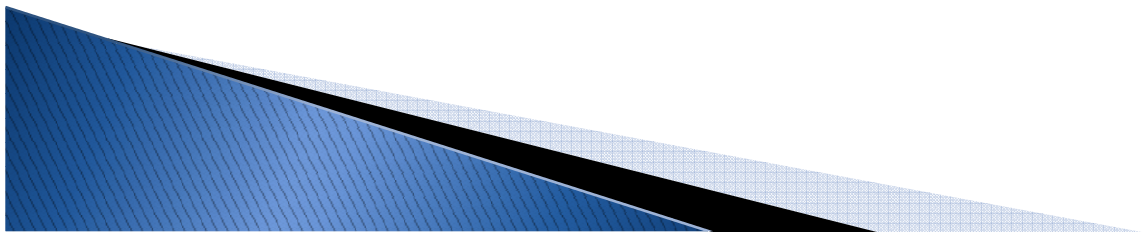
Python Program

- To write a program, put commands in a file

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

- Execute on the command line

```
~> python hello.py
Hello, world
72
```

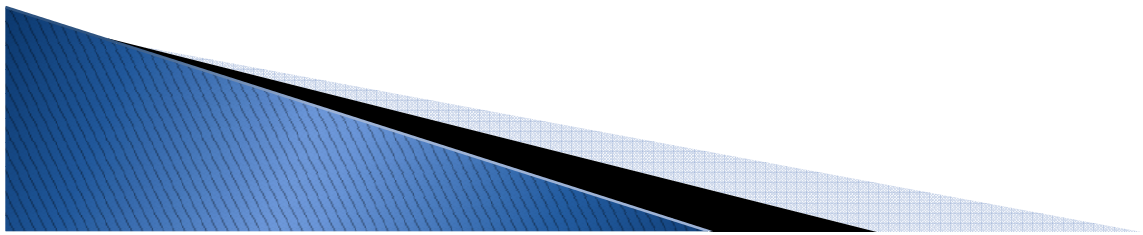


Variables

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

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

- ***Everything*** is an “object":
  - Even functions, classes, modules



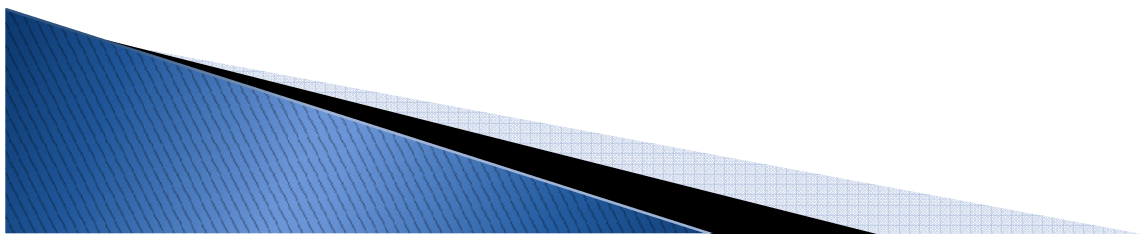
Control Structures

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

while *condition*:
    statements

for *var* in *sequence*:
    statements

break
continue



Grouping Indentation

In Python:

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

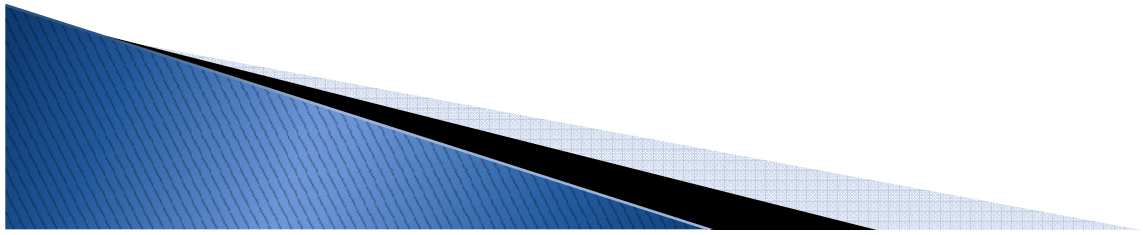
In C:

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

   printf("Bingo!\n"); }
      }
      printf("---\n");
}
```

```
0
Bingo!
---
---
---
3
---
---
---
6
---
---
---
9
---
---
---
12
---
---
---
15
Bingo!
---
---
---
18
---
---
```

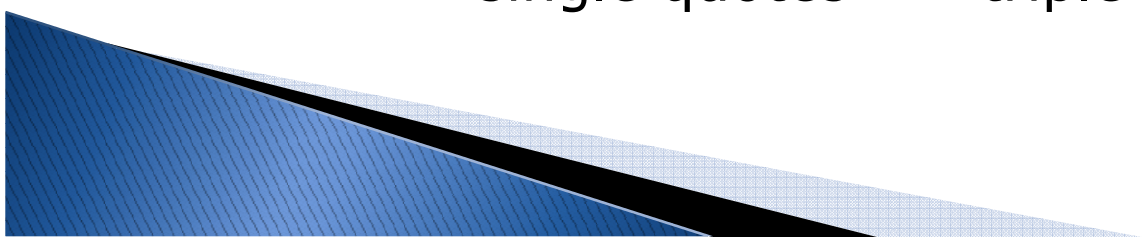
Numbers

- 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 :-(
  - 1/2 -> 0 # 1./2. -> 0.5, float(1)/2 -> 0.5
  - Will be fixed in the future
- Long (arbitrary precision), complex
  - 2L**100 -> 1267650600228229401496703205376L
    - In Python 2.2 and beyond, 2**100 does the same thing
  - 1j**2 -> (-1+0j)



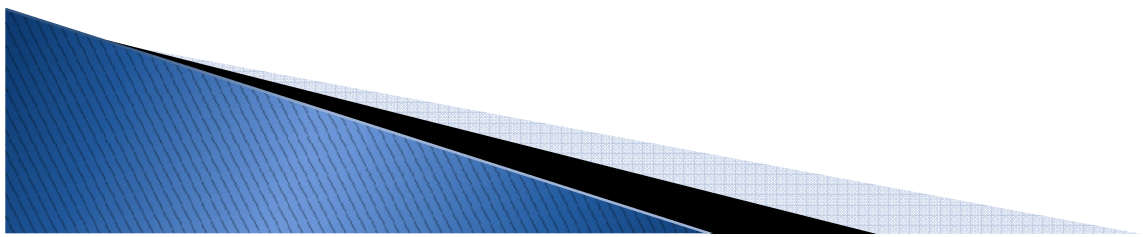
Strings

- "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" 1 # comparison
- "e" in "hello" 1 # search
- "escapes: \n etc, \033 etc, \if etc"
- 'single quotes' """triple quotes""" r"raw strings"



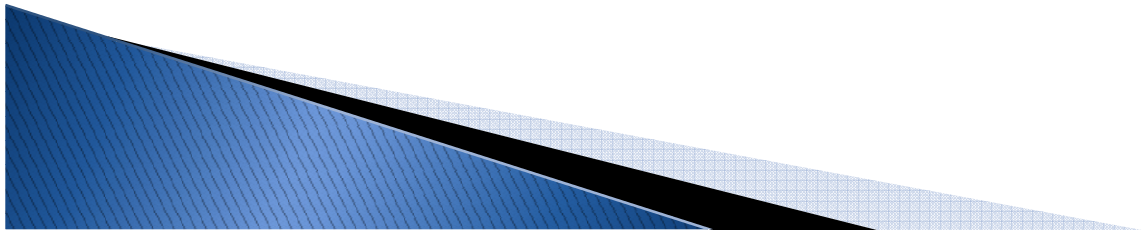
Lists

- Flexible arrays, *not* Lisp-like 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"]



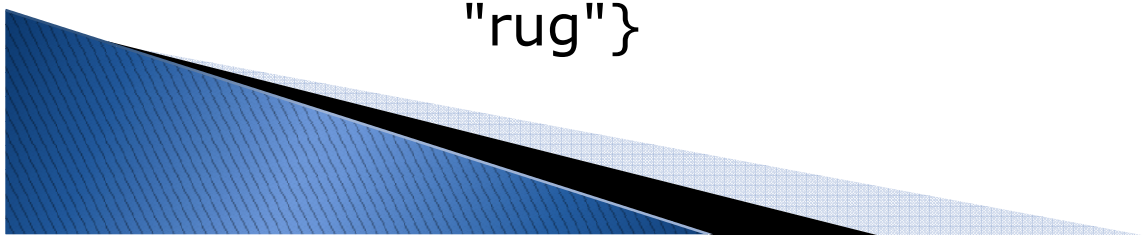
More List Operations

```
>>> 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]
5.5
>>> a.reverse()             # [4,3,2,1,0]
>>> a.sort()                # [0,1,2,3,4]
```



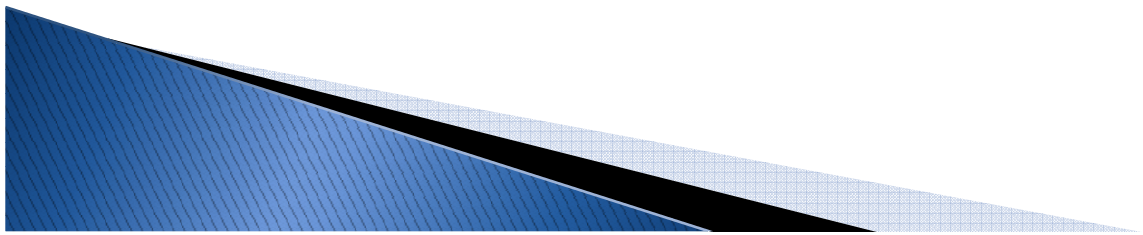
Dictionaries

- 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"}



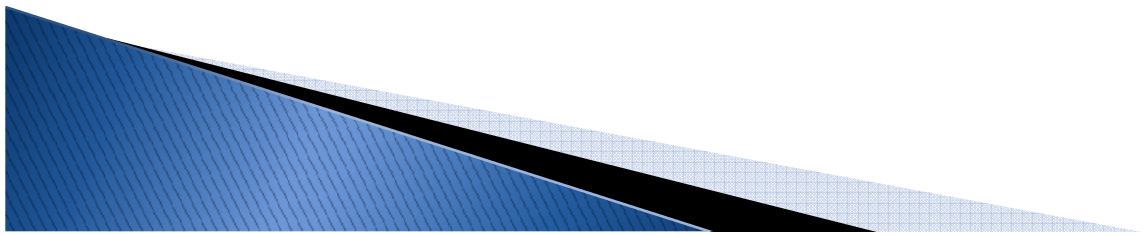
More Dictionary Ops

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



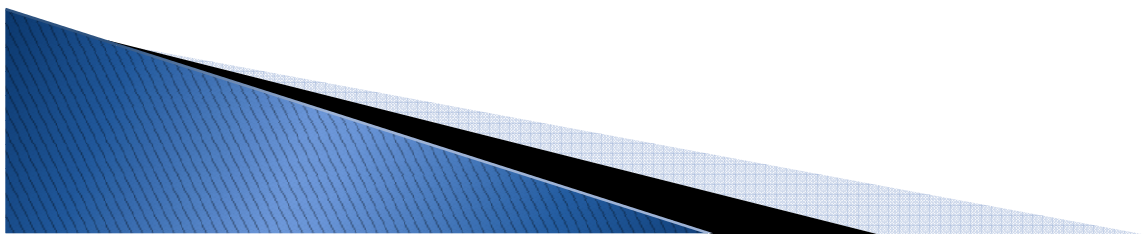
Dictionary Details

- Keys must be **immutable**:
  - numbers, strings, tuples of immutables
    - these cannot be changed after creation
  - reason is *hashing* (fast lookup technique)
  - **not** lists or other dictionaries
    - these types of objects can be changed "in place"
  - no restrictions on values
- Keys will be listed in **arbitrary order**
  - again, because of hashing



Tuples

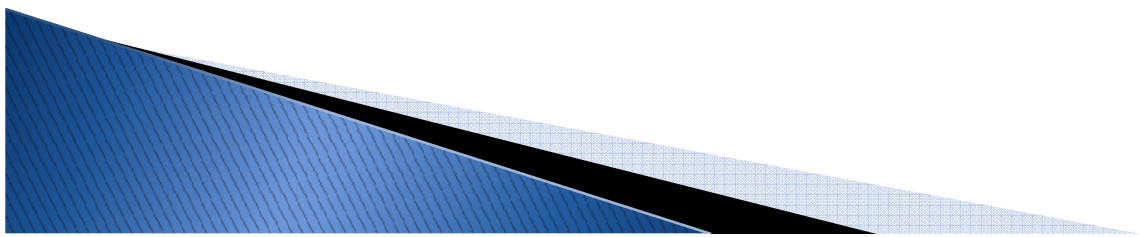
- key = (lastname, firstname)
- point = x, y, z # parentheses optional
- x, y, z = point # unpack
- lastname = key[0]
- singleton = (1,) # trailing comma!!!
- empty = () # parentheses!
- tuples vs. lists; tuples immutable



Reference Semantics

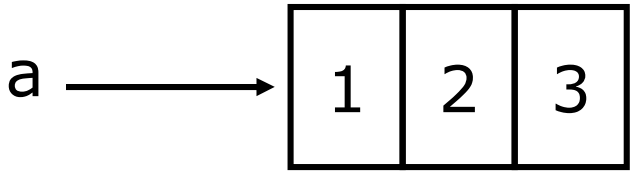
- Assignment manipulates references
  - x = y **does not make a copy** of y
  - x = y makes x **reference** the object y references
- Very useful; but beware!
- Example:

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

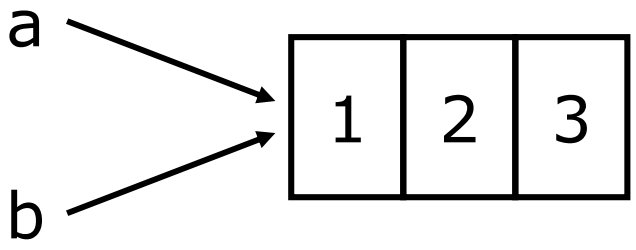


Changing a Shared List

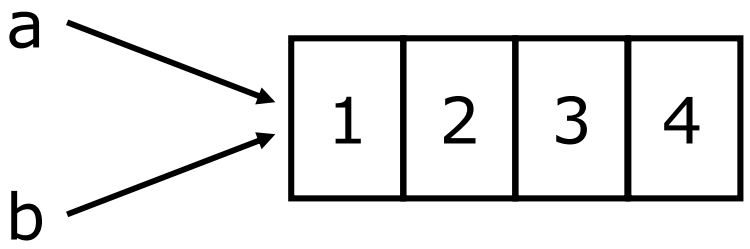
a = [1, 2, 3]



b = a



a.append(4)

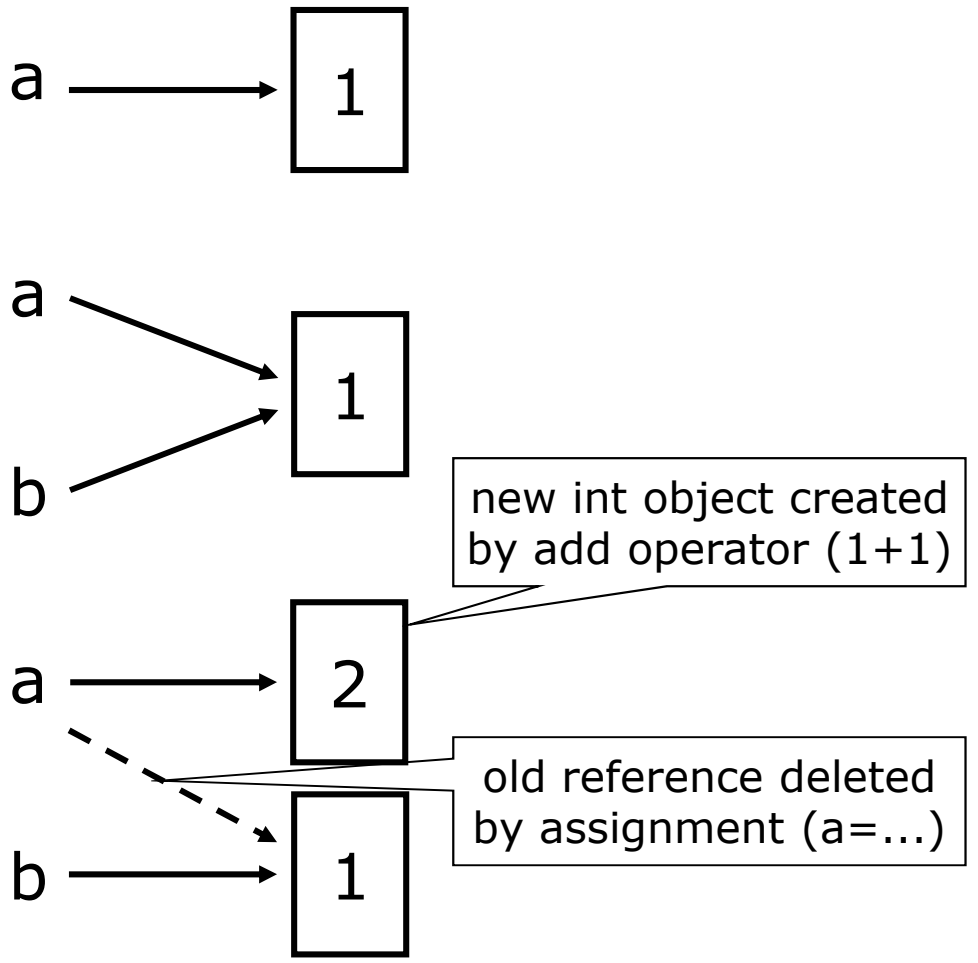


Changing an Integer

a = 1

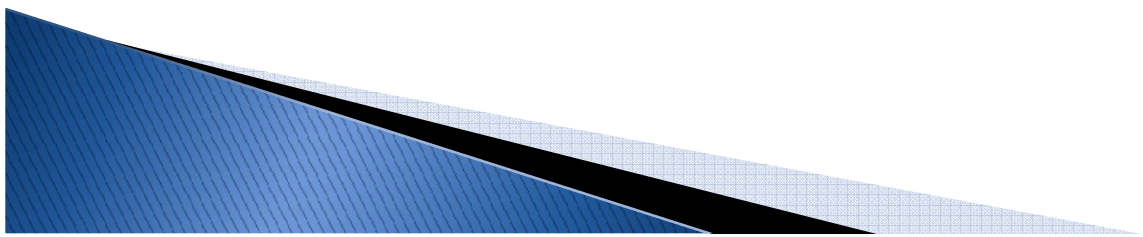
b = a

a = a+1



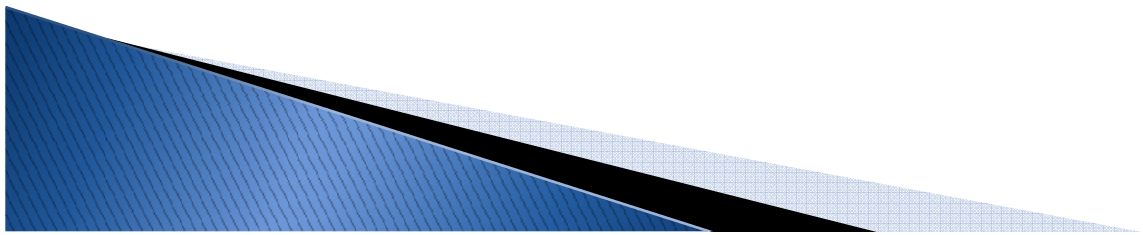
Functions, Procedures

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

return                      # from procedure
return expression                 # from function
```



Example Function

```
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*:

    "*documentation*"

    statements

-or-

class *name*(*base1*, *base2*, ...):

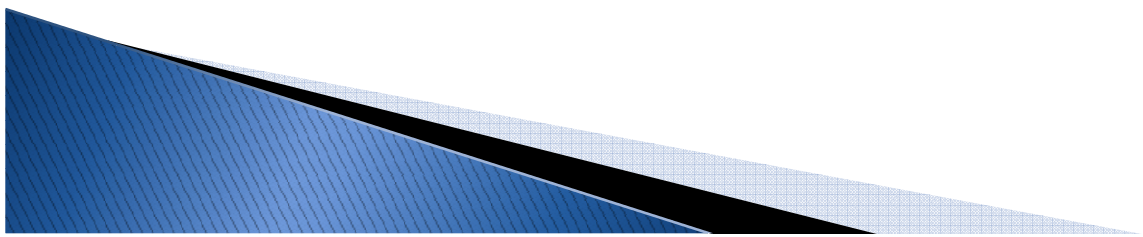
    ...

Most, *statements* are method definitions:

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

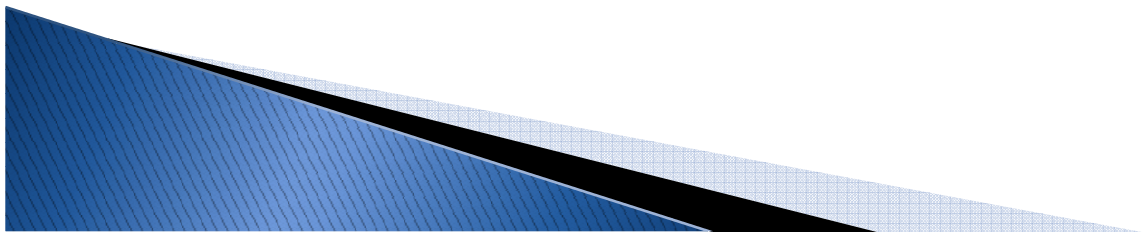
        ...

May also be *class variable* assignments



Example Class

```
class Stack:
    "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 happens if it's empty?
        del self.items[-1]
        return x
    def empty(self):
        return len(self.items) == 0         # Boolean result
```



Using Classes

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

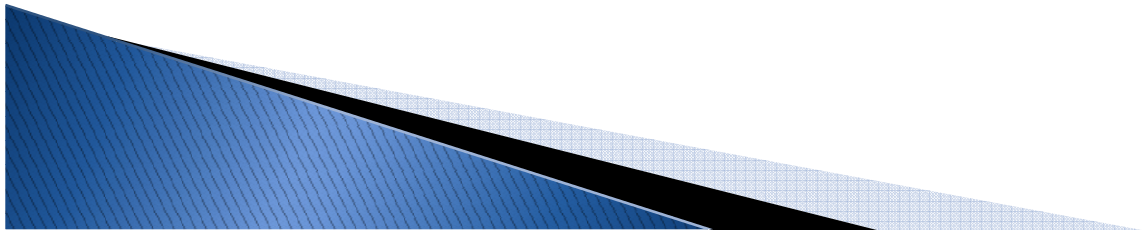
```
x = Stack()# no 'new' operator!
```

- 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]
```

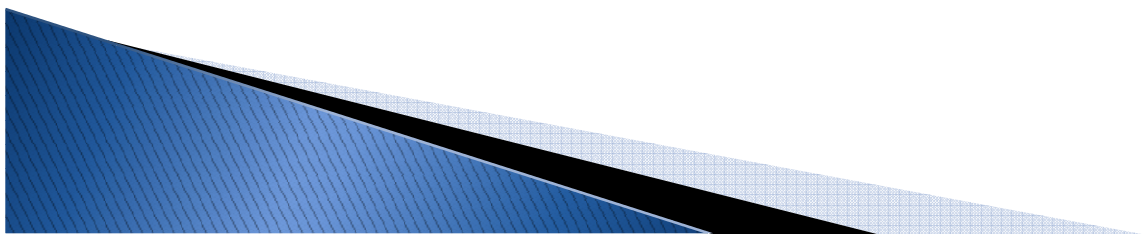
- To inspect instance variables, use dot notation:

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



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



Getting Python

- There are many different flavors of Python, all of which support the same basic API, but have different backends and performance.
- The "original flavor" is CPython, but there is also Jython, Iron Python, Pyjs, PyPy, RubyPython, and others.
- If you are going to use a package with a C extensions, you probably need to get CPython.
- For numerical computational, some additional packages are almost certainly required, NumPy and SciPy being the most obvious.
  - On Linux, Python and the most important packages will be pre-installed, with additional ones installed easily via a package manager.
  - On OS X, Python comes pre-installed, but it is easier to install Python and any additional packages via Homebrew.
  - On Windows, it's easiest to install a distribution that includes the scientific software, such as anaconda or winPython.
- Another option is to use Sage, a Matlab-like collection of Python packages (including COIN).
- Make sure you install Python 2.7!!

Getting an IDE

- An additional requirement for doing development is an IDE.
- My personal choice is Eclipse with the PyDev plug-in.
- This has the advantage of being portable and cross-platform, as well as supporting most major languages.
- There are many alternative IDEs, however.

Exercise: Install Python and IDE

Complete as many of the exercises here as you can:

http://coral.ie.lehigh.edu/~ted/files/ie172/labs/lab0/Lab0.pdf

Make sure you install Python 2.7!!