

# Python – An Introduction

Calle Lejdfors and Lennart Ohlsson

# Overview

- Basic Python
- Numpy



# What is Python?

*Python plays a key role in our production pipeline. Without it a project the size of Star Wars: Episode II would have been very difficult to pull off. From crowd rendering to batch processing to compositing, **Python binds all things together.***

Tommy Burnette, Senior Technical Director, Industrial Light & Magic.

# Conditionals

## If-statements:

```
if x < 0:
    print "negative"
elif x == 0:
    print "zero"
else:
    print "positive"
```

## Logic-ops:

```
if y == 0 or x == 1: ...
if y == 0 and x == 1: ...
if not x == 1: ...
```

Note: Python use indentation delimit blocks *not* { and }.

# Flow control

## For-loops:

```
for animal in ['cat', 'dog', 'bat', 'owl']:  
    print animal
```

## Using range ( )

```
>>> for i in range(4):  
>>>     print i  
0  
1  
2  
3
```

## You (almost) never use:

```
>>> for i in range(len(list)):  
>>>     print list[i]
```

# More on functions

## Doc-strings:

```
>>> def fac( n ):  
...     """computes the factorial of n  
...     if n is negative then fac(n) == 1"""  
...     if n <= 0:  
...         return 1  
...     return n*fac(n-1)  
...     ...
```

```
>>> fac(2)
```

```
2
```

```
>>> help(fac)
```

```
Help on function fac in module __main__:
```

```
fac(n)
```

```
    computes the factorial of n
```

```
    if n is negative then fac(n) == 1
```

# More on functions

## Default arguments:

```
>>> def myfac(n=10): return fac(n)
>>> myfac()
3628800
```

## Keyword-arguments:

```
>>> def f( n=10, m=20, k=3.1415 ) :
...     return (n,m,k)
...
>>> f()
(10, 20, 3.1415000000000002)
>>> f(k=6)
(10, 20, 6)
```

# Numbers

## Integers:

```
>>> 2
2
```

## Floating point number:

```
>>> 3.14159
3.1415899999999999
```

## Complex numbers:

```
>>> 3.1+4.5j
(3.1000000000000001+4.5j)
>>> complex(3.1,4.5)
(3.1000000000000001+4.5j)
```

# Basic Python - Numbers

## Basic arithmetic:

```
>>> 2+3
```

```
5
```

```
>>> 2+3.5
```

```
5.5
```

```
>>> 2+3+4j
```

```
(5+4j)
```

```
>>> 2*3
```

```
6
```

```
>>> 2000000000*20000
```

```
40000000000000L
```

```
>>> 2002/25
```

```
80
```

```
>>> 2002/25.0
```

```
80.079999999999998
```

# Strings

## Different ways to do strings:

```
>>> 'Python test!\n'  
>>> "Python test!\n"  
>>> """Python test!\n"" "  
'Python test!\n'  
>>> "\"Python test\" "  
>>> '"Python test"'  
'"Python test"'
```

## Concatenation:

```
>>> 'Hello' + 'World'  
'HelloWorld'  
>>> 'test'*4  
'testtesttesttest'
```

# Strings

## Length of a string:

```
>>> len(str)
13
```

## Slicing:

```
>>> str = 'A long string'
>>> str[1:]
' long string'
>>> str[:3]
'A l'
>>> str[:-1]
'A long strin'
```

# Functions

A simple factorial function. Computes  $n!$  for any integer  $n$ .

```
def fac( n ):  
    if n <= 0:  
        return 1  
    return n*fac(n-1)
```

# Lists

Lists work like strings but may contain just about anything.

```
>>> aList = [123, 'test', 3.14159, "world" ]
>>> aList[0]
123
>>> aList[2:]
[3.1415899999999999, 'world' ]
```

## Operations on lists:

```
>>> aList.append( '54321' )
>>> aList
[123, 'test', 3.1415899999999999, 'world', '54321' ]
>>> aList.remove( 'test' )
>>> aList
[123, 3.1415899999999999, 'world', '54321' ]
```

# Tuples

A tuple is written as:

```
>>> aTuple = ('first', 2)
>>> aTuple[0]
'first'
>>> aTuple[1]
2
```

Also support tuples without parenthesis:

```
>>> a,b = 1,2
>>> a,b = b,a
>>> a
2
>>> b
1
```

# Dictionaries

Native support for dictionaries.

```
>>> aDict = { 'pi': 3.14159 }
>>> aDict['pi']
3.1415899999999999
>>> aDict.keys()
['pi']
>>> aDict.values()
[3.1415899999999999]
>>> aDict.items()
[('pi', 3.1415899999999999)]
```

Not limited to string keys.

```
>>> aStrangeDict = { 'pi': 3.14159, 4325 : 1.02030 }
>>> aStrangeDict[4325]
1.0203
```

# Functional programming

## Higher-order functions:

```
>>> def fac(n):  
...     if n <= 1: return 1  
...     return n*fac(n-1)  
...  
>>> fac(4)  
24  
  
>>> f = fac # f is now fac  
>>> f(4)  
24
```

Enables us to define functions which take functions as arguments.

# Functional programming

`filter` – remove elements from list:

```
>>> def even(n): return n%2==0
>>> filter( even, range(0,20))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

`map` – apply function to all elements in a list:

```
>>> map(abs, range(-10,10))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

# Functional programming

## Anonymous functions, $\lambda$ -forms

```
>>> addFour = lambda x: x+4
>>> addFour(7)
11
```

## same as

```
>>> def addFour(n): return n+4
>>> addFour(7)
11
```

## Commonly used for small functions

```
>>> filter( lambda x: abs(x) < 5, range(-100,100) )
[-4, -3, -2, -1, 0, 1, 2, 3, 4]
```

# Functional programming

## List comprehensions

```
>>> [x*x for x in range(5)]  
[0,1,4,9,16]  
>>> [x*x for x in range(5) if x%2]  
[1,9]
```

# Modules

## Various ways of importing a module:

```
>>> import os
```

```
>>> help(os.open)
```

Help on built-in function open:

```
open(...)
```

```
open(filename, flag [, mode=0777]) -> fd
```

Open a file (for low level IO).

```
>>> from os import *
```

```
>>> help(open)
```

Help on built-in function open:

```
open(...)
```

```
open(filename, flag [, mode=0777]) -> fd
```

Open a file (for low level IO).

# Classes

## A simple class:

```
class A(object):  
    a = 4  
    b = 5  
  
    def add(self): return self.a+self.b
```

## Some code using this class.

```
>>> a = A()  
>>> print a.add()  
9  
>>> a.a = 14  
>>> print a.add()  
19  
>>> a.add  
<bound method A.add of <__main__.A instance at 0x809d14c>>
```

# Classes

## Constructors

```
class A:
    def __init__(self):
        self.a = 4
        self.b = 5

    def add(self): return self.a+self.b
```

## Inheritance

```
class B(A):
    def __init__(self):
        A.__init__(self)
        self.b = 15
```

```
>>> b = B()
>>> print b.add()
19
```

# Extra: Exceptions

## try-except statement

```
>>> list = range(1,6)
>>> list[8]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>> try:
...     list = range(1,6)
...     list[8]
... except IndexError, e:
...     print 'Got exception:', e
Got exception: list index out of range
```

The catch-all exception is called `Exception`.

```
... except Exception, e:
...     print "Exception '%s' caught!" % (e)
Exception 'list index out of range' caught!
```

# The `global` statement

`global` – What does this code do?

```
>>> t = 2
>>> def f(x): t = x
...
>>> f(3)
>>> print t ## should give 3?
2
```

Since Python implicitly creates a local variable we need:

```
>>> def f(x):
...     global t
...     t = x
...
>>> f(3)
>>> print t ## does return 3
3
```

# Arbitrary arguments

## Anonymous arguments:

```
>>> def f(*args):  
>>>     print args  
>>> f(1,7,4)  
(1,7,4)  
  
>>> def g(**keywords):  
>>>     print keywords  
>>> g(a=3, b='x', y=23423)  
{'a': 3, 'b': 'x', 'y': 23423}
```

`args` is a tuple, `keywords` is a dict

# Numpy

# Numpy

## Fast numerical routines for Python.

```
>>> from numpy import *
>>> a = array( (1,2,3) )
>>> matrix = array( [[1,0,0], [0,1,0], [0,0,1]] )
>>> a
array([1, 2, 3])
>>> matrix
array([[1, 0, 0],
       [0, 1, 0],
       [0, 0, 1]])
```

## Shape

```
>>> a.shape
(3,)
>>> matrix.shape
(3, 3)
```

# Numpy

## Creating filled in arrays

```
>>> zeros((3,3))
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
>>> ones((3,4))
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])
>>> arange(0,10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> identity(4)
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

# Numpy

## Simple operations

```
>>> a = array( [1,2,3] )
>>> a*3
array([3, 6, 9])
>>> a+3
array([4, 5, 6])
>>> b = array( [2,3,4] )
>>> a*b
array([ 2,  6, 12])
>>> b = array( [2.0,3.0,4.0] )
>>> a/b
array([ 0.5          ,  0.66666667,  0.75          ])
```

# Broadcasting

How does Numerics handle operations on arrays of different dimensionality.

## Simple example:

```
>>> a = array( [[1,2], [3,4]] )
>>> b = array( [1] )
>>> print a+b
[[2 3]
 [4 5]]
```

Shapes don't agree so how can we add these arrays?

# Broadcasting

## Another example:

```
>>> a = array( [[1,2,3], [4,5,6]] )
>>> b = array( [1,2,3] )
>>> print a+b
[[2 4 6]
 [5 7 9]]
```

## with shapes

```
>>> print a.shape
(2, 3)
>>> print b.shape
(3,)
```

# Broadcasting

The algorithm:

- Compare the length of each axis starting from the right. Replicate as many times as needed.
- If one of axis compared have length 1 then broadcasting also occur.

**Example:**

```
>>> z = array( [1,2] )
>>> v = array( [[3],[4],[5]] )
>>> print z.shape, v.shape
(2,) (3, 1)
>>> print z+v
[[4 5]
 [5 6]
 [6 7]]
```

# Numpy

Universal functions - functions operating per-element on arrays.

```
>>> a = array( [pi, pi/2, pi/4, pi/8] )
>>> print sin(a)
[ 1.22460635e-16  1.00000000e+00  7.07106781e-01  3.82683432e-01]
```

This works for most standard functions such as

add (+)	subtract (-)	multiply (*)	divide (/)
remainder (%)	power (**)	arccos	arcsin
cos	sin	tan	sqrt
maximum	greater (>)	equal (==)	not_equal (!=)
:			

# Numpy

## reduce – reduce on arrays with ufuncs

```
>>> a = array( [1,2,3,4] )
>>> print add.reduce(a)
10
>>> print multiply.reduce(a)
24
```

## Works for 2d-arrays as well

```
>>> b = array( [ [1,0,0], [0,2,0], [0,0,3] ] )
>>> print add.reduce(b)
[1 2 3]
```

# Numpy

## Reshaping

```
>>> a = arrayrange(1,10)
>>> a.shape=(3,3) ##in-place reshape
>>> a
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b = reshape(a, (3,3)) ## copying reshape
>>> print b
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

# Numpy

## Algebraic operations

```
>>> transpose(a)
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
>>> dot( array([1,2,3]), array([1,0,2]))
7
>>> m = array( [[1,0,0], [0,0,1], [0,1,0]] )
>>> dot(m,array([1,2,3]))
array([1, 3, 2])
```

# Numpy

## Some linear algebra.

```
>>> from numpy.linalg import *
>>> m = array( [[1,2,3], [3,2,1], [5,4,7]] )
>>> inverse(m)
array([[ -0.625,  0.125,  0.25 ],
       [  1.    ,  0.5   , -0.5   ],
       [ -0.125, -0.375,  0.25 ]])
```

# Numpy

## Multi-dimensional slicing

```
>>> a = arange(1,10).reshape((3,3))
```

```
>>> a[0]
```

```
array([1, 2, 3])
```

```
>>> print a[0:2]
```

```
[[1, 2, 3],
```

```
 [4, 5, 6]]
```

```
>>> a[0,2]
```

```
3
```

```
>>> a[0][2]
```

```
3
```

```
>>> print a[0:2,2]
```

```
[3 6]
```

```
>>> print a[0:2,0:2]
```

```
[[1 2]
```

```
 [4 5]]
```

# Numpy

## Reindexing an array

```
>>> a = array( [1,2,3,4] )
>>> i = array( [1,3] )
>>> print a[i]
[2 4]
>>> b = arange(1,10)
>>> b.shape = 3,3
>>> print b
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> print b[array([0,1])]
[[1 2 3]
 [4 5 6]]
```

also possible to use method `take`

# Numpy

## Filter indices for arrays

```
>>> a = array( [[1,2,3], [4,5,6], [7,8,9]] )
>>> b = array( [True, True, False] )
>>> print b[a]
[[1 2 3]
 [4 5 6]]
```

# Numpy

## repeat – replicate array elements

```
>>> a = array( [0,6,-7] )
>>> print a.repeat(2)
[ 0  0  6  6 -7 -7]
>>> b = identity(2)
>>> print b.repeat(3,axis=0)
[[1 0]
 [1 0]
 [1 0]
 [0 1]
 [0 1]
 [0 1]]
```

# Numpy

## concatenate – as for lists

```
>>> a = array( [[1,2,3], [4,5,6], [7,8,9]] )
```

```
>>> concatenate([a,a])
```

```
>>> print concatenate([a,a])
```

```
[[1 2 3]
```

```
[4 5 6]
```

```
[7 8 9]
```

```
[1 2 3]
```

```
[4 5 6]
```

```
[7 8 9]]
```

```
>>> print concatenate([a,a],1)
```

```
[[1 2 3 1 2 3]
```

```
[4 5 6 4 5 6]
```

```
[7 8 9 7 8 9]]
```

# Links

- Python

`http://www.python.org/`

- Tutorial

`http://www.python.org/doc/current/tut.html`

- Numpy

`http://www.pfdubois.com/numpy/`