# Introduction to Scientific Computing with Python

# What Is Python?

Python is an interpreted programming language that allows you to do almost anything possible with a compiled language (C/C++/Fortran) without requiring all the complexity.

## PYTHON HIGHLIGHTS

- **Interpreted and interactive**
- **Object-oriented**
- **"Batteries Included"**
- **Free**
- **Portable**
- **Easy to Learn and Use**
- **Truly Modular**

# Interactive Calculator

```
# adding two values
>>> 1 + 1
2
# setting a variable
>>> a = 1
>>> a
1
# checking a variables type
>>> type(a)
<type 'int'>
# an arbitrarily long integer
>>> a = 1203405503201
>>> a
1203405503201L
>>> type(a)
<type 'long'>
```

```
# real numbers
>>> b = 1.2 + 3.1
>>> b
4.2999999999999998
>>> type(b)
<type 'float'>
# complex numbers
>>> c = 2+1.5j
>>> c
(2+1.5j)
```

# Complex Numbers

## CREATING COMPLEX NUMBERS

```
# Use "j" or "J" for imaginary
# part. Create by "(real+imagj)",
# or "complex(real, imag)" .
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
 (-1+0j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

## EXTRACTING COMPONENTS

```
# to extract real and im
# component
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

# Strings

## CREATING STRINGS

```
# using double quotes
>>> s = "hello world"
>>> print s
hello world
# single quotes also work
>>> s = 'hello world'
>>> print s
hello world
```

## STRING OPERATIONS

```
# concatenating two strings
>>> "hello " + "world"
'hello world'

# repeating a string
>>> "hello " * 3
'hello hello hello '
```

## STRING LENGTH

```
>>> s = "12345"
>>> len(s)
5
```

## FORMAT STRINGS

```
# the % operator allows you
# to supply values to a
# format string.  The format
# string follows
# C conventions.
>>> s = "some numbers:"
>>> x = 1.34
>>> y = 2
>>> s = "%s %f, %d" % (s,x,y)
>>> print s
some numbers: 1.34, 2
```

# Multi-line Strings

```
# triple quotes are used
# for mutli-line strings
>>> a = """hello
... world"""
>>> print a
hello
world

# multi-line strings using
# "\" to indicate
continuation
>>> a = "hello " \
...     "world"
>>> print a
hello world
```

```
# including the new line
>>> a = "hello\n" \
...     "world"
>>> print a
hello
world
```

# List objects

## LIST CREATION WITH BRACKETS

```
>>> l = [10,11,12,13,14]
>>> print l
[10, 11, 12, 13, 14]
```

## CONCATENATING LIST

```
# simply use the + operator
>>> [10, 11] + [12,13]
[10, 11, 12, 13]
```

## REPEATING ELEMENTS IN LISTS

```
# the multiply operator
# does the trick.
>>> [10, 11] * 3
[10, 11, 10, 11, 10, 11]
```

## range( start, stop, step)

```
# the range method is helpful
# for creating a sequence
>>> range(5)
[0, 1, 2, 3, 4]

>>> range(2,7)
[2, 3, 4, 5, 6]

>>> range(2,7,2)
[2, 4, 6]
```

# Indexing

## RETREIVING AN ELEMENT

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

## SETTING AN ELEMENT

```
>>> l[1] = 21
>>> print l
[10, 21, 12, 13, 14]
```

## OUT OF BOUNDS

```
>>> l[10]
Traceback (innermost last):
File "<interactive input>",line 1,in ?
IndexError: list index out of range
```

## NEGATIVE INDICES

```
# 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
```

# More on list objects

## LIST CONTAINING MULTIPLE TYPES

```
# list containing integer,
# string, and another list.
>>> l = [10,'eleven',[12,13]]
>>> l[1]
'eleven'
>>> l[2]
[12, 13]

# use multiple indices to
# retrieve elements from
# nested lists.
>>> l[2][0]
12
```

## LENGTH OF A LIST

```
>>> len(l)
3
```

## DELETING OBJECT FROM LIST

```
# use the del keyword
>>> del l[2]
>>> l
[10,'eleven']
```

## DOES THE LIST CONTAIN x ?

```
# use in or not in
>>> l = [10,11,12,13,14]
>>> 13 in l
1
>>> 13 not in l
0
```

# Slicing

**var[lower:upper]**

## SLICING LISTS

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

# negative indices work also
>>> l[1:-2]
[11, 12]
>>> l[-4:3]
[11, 12]
```

## OMITTING INDICES

```
# omitted boundaries are
# assumed to be the beginning
# (or end) of the list.

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

# List methods in action

```
>>> l = [10,21,23,11,24]

# add an element to the list
>>> l.append(11)
>>> print l
[10,21,23,11,24,11]

# how many 11s are there?
>>> l.count(11)
2


# where does 11 first occur?
>>> l.index(11)
3
```

```
# remove the first 11
>>> l.remove(11)
>>> print l
[10,21,23,24,11]

# sort the list
>>> l.sort()
>>> print l
[10,11,21,23,24]

# reverse the list
>>> l.reverse()
>>> print l
[24,23,21,11,10]
```

# Dictionaries

Dictionaries store *key/value* pairs. Indexing a dictionary by a *key* returns the *value* associated with it.

**DICTIONARY EXAMPLE**

```
# create an empty dictionary using curly brackets
>>> record = {}
>>> record[‘first’] = ‘Jmes’
>>> record[‘last’] = ‘Maxwell’
>>> record[‘born’] = 1831
>>> print record
{'first': 'Jmes', 'born': 1831, 'last': 'Maxwell'}
```

# Dictionary methods in action

```
>>> d = {'cows': 1,'dogs':5,
...       'cats': 3}
```

```
# get a list of all keys
>>> d.keys()
['cats','dogs','cows']
```

```
# get a list of all values
>>> d.values()
[3, 5, 1]
```

```
# clear the dictionary
>>> d.clear()
>>> print d
{}
```

# If statements

`if/elif/else` provide conditional execution of code blocks.

## IF STATEMENT FORMAT

```
if <condition>:
        <statements>
elif <condition>:
        <statements>
else:
        <statements>
```

## IF EXAMPLE

```
# a simple if statement
>>> x = 10
>>> if x > 0:
...     print 1
... elif x == 0:
...     print 0
... else:
...     print -1
... < hit return >
1
```

# Test Values

- True means any non-zero number or non-empty object
- False means not true: zero, empty object, or **`None`**

**EMPTY OBJECTS**

```
# empty objects evaluate false
>>> x = []
>>> if x:
...     print 1
... else:
...     print 0
... < hit return >
0
```

# For loops

For loops iterate over a sequence of objects.

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

**TYPICAL SCENARIO**

```
>>> for i in range(5):
...     print i,
... < hit return >
0 1 2 3 4
```

**LOOPING OVER A STRING**

```
>>> for i in 'abcde':
...     print i,
... < hit return >
a b c d e
```

**LOOPING OVER A LIST**

```
>>> l=['dogs','cats','bears']
>>> accum = ''
>>> for item in l:
...     accum = accum + item
...     accum = accum + ' '
... < hit return >
>>> print accum
dogs cats bears
```

# While loops

While loops iterate until a condition is met.

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

## WHILE LOOP

```
# the condition tested is
# whether lst is empty.
>>> lst = range(3)
>>> while lst:
...     print lst
...     lst = lst[1:]
... < hit return >
[0, 1, 2]
[1, 2]
[2]
```

## BREAKING OUT OF A LOOP

```
# breaking from an infinite
# loop.
>>> i = 0
>>> while 1:
...     if i < 3:
...         print i,
...     else:
...         break
...     i = i + 1
... < hit return >
0 1 2
```

# Anatomy of a function

The keyword **`def`** indicates the start of a function.

Function arguments are listed separated by commas. They are passed by *assignment*. More on this later.

```
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`**.

# Our new function in action

```
# We'll create our function
# on the fly in the
# interpreter.
>>> def add(x,y):
...     a = x + y
...     return a

# test it out with numbers
>>> x = 2
>>> y = 3
>>> add(x,y)
5
```

```
# how about strings?
>>> x = 'foo'
>>> y = 'bar'
>>> add(x,y)
'foobar'

# functions can be assigned
# to variables
>>> func = add
>>> func(x,y)
'foobar'
```

```
# how about numbers and strings?
>>> add('abc',1)
Traceback (innermost last):
File "<interactive input>", line 1, in ?
File "<interactive input>", line 2, in add
TypeError: cannot add type "int" to string
```

# Modules

## EX1.PY

```
# ex1.py

PI = 3.1416

def sum(lst):
    tot = lst[0]
    for value in lst[1:]:
        tot = tot + value
    return tot

l = [0,1,2,3]
print sum(l), PI
```

## FROM SHELL

```
[ej@bull ej]$ python ex1.py
6, 3.1416
```

# NumPy

# NumPy

## IMPORT NUMPY

```
>>> from numpy import *
>>> import numpy
```

# Array Operations

## SIMPLE ARRAY MATH

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

```
# multiply entire array by
# scalar value
>>> a = (2*pi)/10.
>>> a
0.628318530718
```

## MATH FUNCTIONS

```
>>> a*x
array([ 0.,0.628,…,6.283])
```

```
# apply functions to array.
>>> y = sin(a*x)
```

# Introducing Numeric Arrays

## SIMPLE ARRAY CREATION

```
>>> a = array([0,1,2,3])
>>> a
array([0, 1, 2, 3])
```

## CHECKING THE TYPE

```
>>> type(a)
<type 'array'>
```

## NUMERIC TYPE OF ELEMENTS

```
>>> a.typecode()
'l`    # 'l' = Int
```

## BYTES IN AN ARRAY ELEMENT

```
>>>
a.itemsize()
4
```

## ARRAY SHAPE

```
>>> a.shape
(4,)
>>> shape(a)
(4,)
```

## CONVERT TO PYTHON LIST

```
>>> a.tolist()
[0, 1, 2, 3]
```

## ARRAY INDEXING

```
>>> a[0]
0
>>> a[0] = 10
>>> a
[10, 1, 2, 3]
```

# Multi-Dimensional Arrays

## MULTI-DIMENSIONAL ARRAYS

```
>>> a = array([[ 0, 1, 2, 3],
               [10,11,12,13]])
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,13]])
```

## (ROWS,COLUMNS)

```
>>> shape(a)
(2, 4)
```

## GET/SET ELEMENTS

```
>>> a[1,3]
13
```

column

row

```
>>> a[1,3] = -1
>>> a
array([[ 0, 1, 2, 3],
       [10,11,12,-1]])
```

## ADDRESS FIRST ROW USING SINGLE INDEX

```
>>> a[1]
array([10, 11, 12, 13])
```

## FLATTEN TO 1D ARRAY

```
>>> a.flat
array(0,1,2,3,10,11,12,-1)
```

# Array Slicing

**SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING**

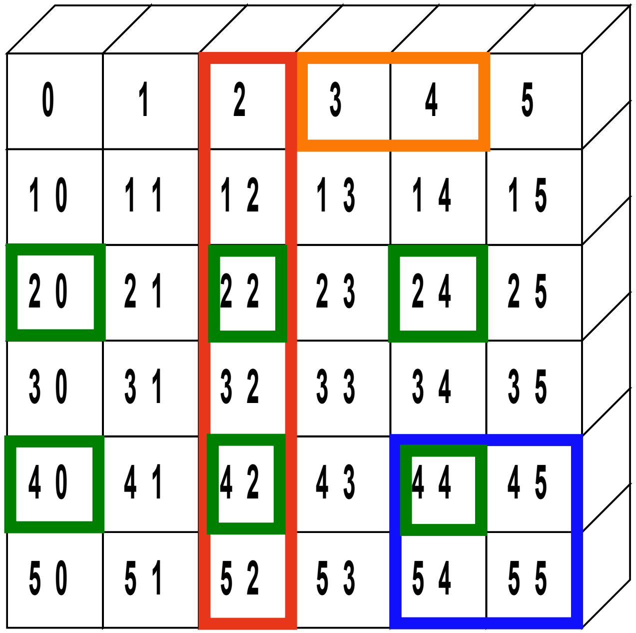
```
>>> a[0,3:5]
array([3, 4])

>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])
```

**STRIDES ARE ALSO POSSIBLE**

```
>>> a[2::2,::2]
array([[20, 22, 24],
       [40, 42, 44]])
```



# Mathematic Binary Operators

```
a + b →  add(a,b)
a - b →  subtract(a,b)
a % b →  remainder(a,b)
```

```
a * b  →  multiply(a,b)
a / b  →  divide(a,b)
a ** b →  power(a,b)
```

## MULTIPLY BY A SCALAR

```
>>> a = array((1,2))
>>> a*3.
array([3., 6.])
```

## ELEMENT BY ELEMENT ADDITION

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

## ADDITION USING AN OPERATOR FUNCTION

```
>>> add(a,b)
array([4, 6])
```

# Comparison and Logical Operators

| | | | | | |
|---|---|---|---|---|---|
| `equal` | `(==)` | `not_equal` | `(!=)` | `greater` | `(>)` |
| `greater_equal` | `(>=)` | `less` | `(<)` | `less_equal` | `(<=)` |
| `logical_and` | `(and)` | `logical_or` | `(or)` | `logical_xor` | |
| `logical_not` | `(not)` | | | | |

## 2D EXAMPLE

```
>>> a = array(((1,2,3,4),(2,3,4,5)))
>>> b = array(((1,2,5,4),(1,3,4,5)))
>>> a == b
array([[1, 1, 0, 1],
       [0, 1, 1, 1]])
# functional equivalent
>>> equal(a,b)
array([[1, 1, 0, 1],
       [0, 1, 1, 1]])
```