Computational Physics

Objects : Lists & Arrays

Prof. Paul Eugenio
Department of Physics
Florida State University
Jan 24, 2019

http://hadron.physics.fsu.edu/~eugenio/comphy/

Announcements

Read chapter 3

Graphics and visualization

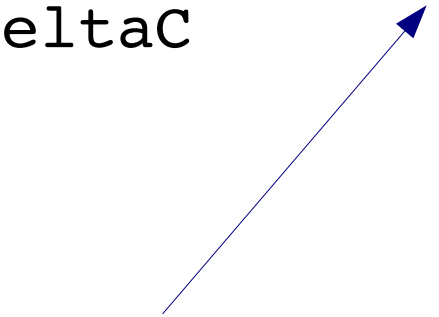
[No turn in questions]

while loop

```
…

C = -20
deltaC = 5

print(“C\tF”)
while C <= 40:
    F = (9/5)*C + 32
    print(C, F, sep='\t')
    C += deltaC
```




```
physics 654% cel2fah.py
 C      F
-20    -4.0
-15    5.0
-10    14.0
-5     23.0
0      32.0
5      41.0
10     50.0
15     59.0
20     68.0
25     77.0
30     86.0
35     95.0
40     104.0
```

Object list

- A list is a basic container for storing data elements

```
>>> C = [-10, 0, 10, 20, 30, 40]
>>> print(C)
[-10, 0, 10, 20, 30, 40]
```

- A list can contain different types

```
>>> M = [-10, 14.0, “cold”]
>>> print(M)
[-10, 14.0, 'cold']
```

The *list* container

- Adding elements to a list

```
>>> A = [10, 20]
>>> A += [30]          # same as A.append(30)
>>> A = A + [40,30]
>>> print(A)
[10, 20, 30, 40, 30]
```

- Functions operating on lists

```
>>> mean = sum(A)/len(A)
>>> print(mean)
26
>>> print(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

More on *range()*

- `range()` creates a list of a given length

```
>>> range(5)
[1, 2, 3, 4]

>>> range(2, 8)
[2, 3, 4, 5, 6, 7]

>>> range(2, 20, 3)
[2, 5, 8, 11, 14, 17]

>>> range(20, 2, -3)
[20, 17, 14, 11, 8, 5]
```

- Often used with a `for` loop

for loop with a `list`

```
…
C = [-10, 0, 10, 20, 30, 40]

print("C\t\tF\t\tFw\t\tFs")

for tempC in C:
    F = (9/5)*tempC + 32
    Fw = 2*C + 30
    Fs = 2*C + 25
    print(C, F, Fw, Fs, sep='\t\t')
```

tempC will iterate through the list C values

for loop with a `list`

```
…
C = [-10, 0, 10, 20, 30, 40]

print("C\t\tF\t\tFw\t\tFs")

for tempC in C:
    F = (9/5)*tempC + 32
    Fw = 2*C + 30
    Fs = 2*C + 25
    print(C, F, Fw, Fs, sep='\t\t')
```

tempC will iterate through the list C values

C	F	Fw	Fs
-10	14.0	10	5
0	32.0	30	25
10	50.0	50	45
20	68.0	70	65
30	86.0	90	85
40	104.0	110	105

winter

summer

SciPy.org

SciPy is a Python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, these are some of the core packages:

NumPy
Base N-dimensional array package

SciPy library
Fundamental library for scientific computing

Matplotlib
Comprehensive 2D Plotting

IP[y]: IPython

IPython
Enhanced Interactive Console

Sympy
Symbolic mathematics

pandas
Data structures & analysis

NumPy [numpy.org]

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

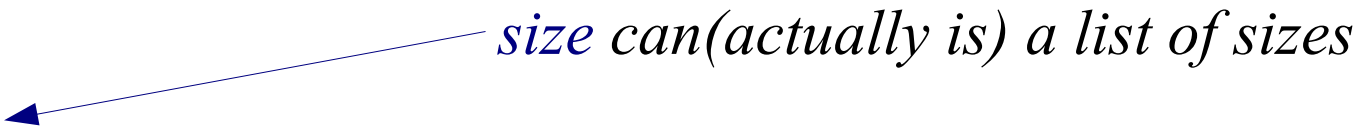
- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

The numpy `array` container

- The number of elements in an array is fixed
  - no resize of array
- Array elements must be of the same type
  - must be all floats, all ints, all complex, ...
- Arrays can be multidimensional
  - like matrices in algebra
- Arrays behave like vectors or matrices
  - you can do arithmetic with them

numpy *array*



- zeros(size, type)
  - create an array with all elements zero
- empty(size, type)
  - create an array without initializing elements
- array(list, type)
  - create an array using a list of values

numpy *array*

Python interactively

```
hpc-login 674% python

>>> import numpy as np
>>>
>>> np.zeros(2)
array([ 0.,  0.])

>>> np.zeros([2,2], int)
array([[ 0,  0],
       [ 0,  0]])
```

numpy *array*

Python interactively

```
hpc-login 674% python

>>> import numpy as np
>>>
>>> np.zeros(2)
array([ 0.,  0.])

>>> np.zeros([2,2], int)
array([[ 0,  0],
       [ 0,  0]])
```

Python code

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

#...
import numpy as np

print(np.zeros(2))
print(np.zeros([2,2], int)
```

```
hpc-login 675% chmod +x testArrays.py
hpc-login 676% testArrays.py

[ 0.  0.]

[[ 0  0]
 [ 0  0]]
```

Complex Numbers

In python, you can put ‘j’ after a number to make it imaginary

$$\sqrt{-1} = 1j$$

```
>>> j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined
```

Complex Numbers

In python, you can put ‘j’ after a number to make it imaginary

$$\sqrt{-1} = 1j$$

```
>>> j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined

>>> 1j
1j
```

Complex Numbers

In python, you can put ‘j’ after a number to make it imaginary

$$\sqrt{-1} = 1j$$

```
>>> j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined

>>> 1j
1j
>>> 1j * 1j
(-1+0j)
```

Complex Numbers

In python, you can put ‘j’ after a number to make it imaginary

$$\sqrt{-1} = 1j$$

```
>>> j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined

>>> 1j
1j
>>> 1j * 1j
(-1+0j)
>>> complex(2, 3)  #The data type of a complex number is complex
(2+3j)
>>> z = 2+3j
```

Complex Numbers

In python, you can put ‘j’ after a number to make it imaginary

$$\sqrt{-1} = 1j$$

```
>>> j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined

>>> 1j
1j
>>> 1j * 1j
(-1+0j)
>>> complex(2, 3)  #The data type of a complex number is complex
(2+3j)
>>> z = 2+3j
>>> z.real
2.0
>>> z.imag
3.0
```

Complex Numbers

In python, you can put ‘j’ after a number to make it imaginary

$$\sqrt{-1} = 1j$$

```
>>> j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined

>>> 1j
1j
>>> 1j * 1j
(-1+0j)
>>> complex(2, 3)  #The data type of a complex number is complex
(2+3j)
>>> z = 2+3j
>>> z.real
2.0
>>> z.imag
3.0
>>> z.conjugate()   #same as: complex.conjugate(z)
(2-3j)
```

Complex Numbers

In python, you can put ‘j’ after a number to make it imaginary

$$\sqrt{-1} = 1j$$

```
>>> j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined

>>> 1j
1j
>>> 1j * 1j
(-1+0j)
>>> complex(2, 3)  #The data type of a complex number is complex
(2+3j)
>>> z = 2+3j
>>> z.real
2.0
>>> z.imag
3.0
>>> z.conjugate()   #same as: complex.conjugate(z)
(2-3j)
>>> z*z
(-5+12j)
>>> z*z.conjugate()
(13+0j)
```

Complex Numbers

In python, you can put 'j' after a number to make it imaginary

$$\sqrt{-1} = 1j$$

```
>>> j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'j' is not defined

>>> 1j
1j
>>> 1j * 1j
(-1+0j)
>>> complex(2, 3)  #The data type of a complex number is complex
(2+3j)
>>> z = 2+3j
>>> z.real
2.0
>>> z.imag
3.0
>>> z.conjugate()   #same as: complex.conjugate(z)
(2-3j)
>>> z*z
(-5+12j)
>>> z*z.conjugate()
(13+0j)
```

The module ***cmath*** has more functions that handle complex numbers

```
>>> import cmath
>>> cmath.sin(2 + 3j)
(9.154499-4.168906j)
```

Pauli matrices

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$\sigma_x^2 = \sigma_y^2 = \sigma_z^2 = -i\sigma_x\sigma_y\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$

Pauli matrices

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$\sigma_x^2 = \sigma_y^2 = \sigma_z^2 = -i\sigma_x\sigma_y\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$

Python interactively

```
>>> import numpy as np
>>> pauliX = np.array( [[0, 1], [1, 0]], complex )
>>> pauliY = np.array( [[0, -1j], [1j, 0]], complex )
>>> pauliZ = np.array( [[1, 0], [0, -1]], complex )

>>> pauliX
array([[ 0.+0.j,  1.+0.j],
       [ 1.+0.j,  0.+0.j]])

>>> pauliY
array([[ 0.+0.j,  0.-1.j],
       [ 0.+1.j,  0.+0.j]])

>>> pauliZ
array([[ 1.+0.j,  0.+0.j],
       [ 0.+0.j, -1.+0.j]])
```

Pauli matrices

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

$$\sigma_x^2 = \sigma_y^2 = \sigma_z^2 = -i\sigma_x\sigma_y\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$$

Python interactively

```
>>> import numpy as np
>>> pauliX = np.array( [[0, 1], [1, 0]], complex )
>>> pauliY = np.array( [[0, -1j], [1j, 0]], complex )
>>> pauliZ = np.array( [[1, 0], [0, -1]], complex )

>>> pauliX
array([[ 0.+0.j,  1.+0.j],
       [ 1.+0.j,  0.+0.j]])

>>> pauliY
array([[ 0.+0.j,  0.-1.j],
       [ 0.+1.j,  0.+0.j]])

>>> pauliZ
array([[ 1.+0.j,  0.+0.j],
       [ 0.+0.j, -1.+0.j]])
```

pauliX.dot(pauliX)

Object-oriented equivalent

```
>>> np.dot( pauliX, pauliX )
array([[ 1.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j]])

>>> pauliY.dot( pauliY )
array([[ 1.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j]])

>>> pauliZ.dot( pauliZ )
array([[ 1.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j]])

>>> -1j*pauliX.dot( pauliY ).dot( pauliZ )
array([[ 1.+0.j,  0.+0.j],
       [ 0.+0.j,  1.+0.j]])
```

Solving Systems of Linear Equations

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Solving Systems of Linear Equations

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

multiply both sides by the **inverse** of the **matrix *c***

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}^{-1} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}^{-1} \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}^{-1} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Solving Systems of Linear Equations

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

multiply both sides by the inverse of the matrix ***c***

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}^{-1} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}^{-1} \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \Rightarrow \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}^{-1} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Example

Solve for x and y from:

$$\begin{aligned} 1x + 3y &= -1 \\ 2x + 4y &= 3 \end{aligned}$$

```
# matrixInversion.py
from __future__ import division, print_function
import numpy as np
import scipy.linalg as sla

a = np.array( [-1, 3] )
c = np.array( [ [1, 3], [2, 4] ] )
x = np.dot( sla.inv(c), a )          # or use x = sla.inv(c).dot(a)
print("Results: x =", x[0], "and y =", x[1] )
```

Solving Systems of Linear Equations

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

multiply both sides by the inverse of the matrix ***c***

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}^{-1} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}^{-1} \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \Rightarrow \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}^{-1} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Example

Solve for x and y from:

$$\begin{aligned} 1x + 3y &= -1 \\ 2x + 4y &= 3 \end{aligned}$$

```
# matrixInversion.py
from __future__ import division, print_function
import numpy as np
import scipy.linalg as sla

a = np.array( [-1, 3] )
c = np.array( [ [1, 3], [2, 4] ] )
x = np.dot( sla.inv(c), a )          # or use x = sla.inv(c).dot(a)
print("Results: x =", x[0], "and y =", x[1] )
```

```
hpc-login 672% matrixInversion.py
Results: x = 6.5 and y = -2.5
```

Slicing lists and arrays

- One can create a sub list(or array) from a list(or array)

if r is a list then r[m,n] is a sub list starting with element m going up to but not including element n

```
r = [1, 3, 5, 7, 9, 11, 13, 15]
```

Slicing lists and arrays

- One can create a sub list(or array) from a list(or array)

if r is a list then r[m,n] is a sub list starting with element m going up to but not including element n

```
      0, 1, 2, 3, 4, 5, 6, 7, 9
r = [1, 3, 5, 7, 9, 11, 13, 15]

s = r[2:5]
print(s)              # prints [5, 7, 9]
print(r[:3])          # prints [1, 3, 5]
print(r[3:])          # prints [7, 9, 11, 13, 15]
```

Recursive Functions

A function which calls itself

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

# recursive.py Recursive Function: counts down
#
from __future__ import division, print_function

# Recursively count down
def recursion(n):
    if n>=0 :
        print(n)
        recursion(n-1)

recursion(5)
```

function is calling itself!

Recursive Functions

A function which calls itself

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

# recursive.py Recursive Function: counts down
#
from __future__ import division, print_function

# Recursively count down
def recursion(n):
    if n>=0 :
        print(n)
        recursion(n-1)

recursion(5)
```

function is calling itself!

```
hpc-login 672% recursive.py
5
4
3
2
1
0
```

Recursive Functions

Counting up via `return`

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

# recursiveUp.py Recursive Function: counts up
#
from __future__ import division, print_function

# Recursively count up
def recursionUp(n):
    if n>=0 :
        print(1 + recursionUp(n-1))
    return n

recursionUp(5)
```

Recursive Functions

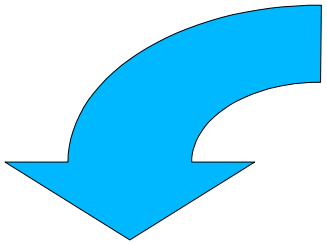
Counting up via `return`

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

# recursiveUp.py Recursive Function: counts up
#
from __future__ import division, print_function

# Recursively count up
def recursionUp(n):
    if n>=0 :
        print(1 + recursionUp(n-1))
    return n

recursionUp(5)
```



```
hpc-login 672% recursiveUp.py
0
1
2
3
4
5
```

Let's get working