Introduction to

Scientific Python programming

Apdapted to TKT4140 Numerical Methods with Computer Laboratory

Hans Petter Langtangen[1,2] (**hpl@simula.no**)

Leif Rune Hellevik[3] (**leif.r.hellevik@ntnu.no**)

[1]Simula Research Laboratory
[2]University of Oslo
[3]Biomechanichs Group, Departement of Structural Engineering NTNU

Jan 12, 2015

Contents

This is a very quick intro to Python programming

- variables for numbers, lists, and arrays
- while loops and for loops
- functions
- if tests
- plotting

Method: show program code through math examples

Variables, loops, lists, and arrays

Mathematical example

Most examples will involve this formula:

$$s = v_0 t + \frac{1}{2} a t^2$$

We may view s as a function of t: $s(t)$, and also include the parameters in the notation: $s(t; v_0, a)$.

A program for evaluating a formula

Task. Compute s for $t = 0.5$, $v_0 = 2$, and $a = 0.2$.

Code.

```
t = 0.5
v0 = 2
a = 0.2
s = v0*t + 0.5*a*t**2
print s
```

Execution.

```
Terminal> python distance.py
1.025
```

Assignment statements assign a name to an object

```
t = 0.5                  # real number makes float object
v0 = 2                   # integer makes int object
a = 0.2                  # float object
s = v0*t + 0.5*a*t**2    # float object
```

Rule: evaluate right-hand side object, left-hand side is a name for that object

Formatted output with text and numbers

- Task: write out `s=1.025`
- Method: printf syntax

```
print 's=%g' % s       # g: compact notation
print 's=%.2f' % s     # f: decimal notation, .2f: 2 decimals
```

Modern alternative: format string syntax

```
print 's={s:.2f}'.format(s=s)
```

Programming with a while loop

- Task: write out a table of t and $s(t)$ values (two columns), for $t \in [0, 2]$ in steps of 0.1
- Method: while loop

```
v0 = 2
a = 0.2
dt = 0.1  # Increment
t = 0     # Start value
while t <= 2:
    s = v0*t + 0.5*a*t**2
    print t, s
    t = t + dt
```

Output of the previous program

```
Terminal> python while.py
0 0.0
0.1 0.201
0.2 0.404
0.3 0.609
0.4 0.816
0.5 1.025
0.6 1.236
0.7 1.449
0.8 1.664
0.9 1.881
1.0 2.1
```

```
1.1 2.321
1.2 2.544
1.3 2.769
1.4 2.996
1.5 3.225
1.6 3.456
1.7 3.689
1.8 3.924
1.9 4.161
```

Structure of a while loop

```
while condition:
    <intented statement>
    <intented statement>
    <intented statement>
```

Note:

- the colon in the first line
- all statements in the loop must be indented
- `condition` is a boolean expression (e.g., `t <= 2`)

The Python Online Tutor can help you understand the program flow

Python Online Tutor lets you step through the program and examine variables.

```
a = 1
da = 0.5
while a <= 3:
    print a
    a = a + da
```

(Visualize execution)

Lists

A list collects several variables (objects) in a given sequence:

```
L = [-1, 1, 8.0]
```

A list can contain any type of objects, e.g.,

```
L = ['mydata.txt', 3.14, 10]
```

Some basic list operations:

```
>>> L = ['mydata.txt', 3.14, 10]
>>> print L[0]
mydata.txt
>>> print L[1]
3.14
>>> del L[0]  # delete the first element
>>> print L
[3.14, 10]
>>> print len(L)  # length of L
2
>>> L.append(-1)  # add -1 at the end of the list
>>> print L
[3.14, 10, -1]
```

Store our table in two lists, one for each column

```
v0 = 2
a = 0.2
dt = 0.1  # Increment
t = 0
t_values = []
s_values = []
while t <= 2:
    s = v0*t + 0.5*a*t**2
    t_values.append(t)
    s_values.append(s)
    t = t + dt
print s_values  # Just take a look at a created list

# Print a nicely formatted table
i = 0
while i <= len(t_values)-1:
    print '%.2f  %.4f' % (t_values[i], s_values[i])
    i += 1   # Same as i = i + 1
```

For loops

A for loop is used for visiting elements in a list, one by one:

```
>>> L = [1, 4, 8, 9]
>>> for e in L:
...     print e
...
1
4
8
9
```

Demo in the Python Online Tutor:

```
list1 = [0, 0.1, 0.2]
list2 = []
for element in list1:
    p = element + 2
    list2.append(p)
print list2
```

(Visualize execution)

For loops used traditionally an integer counter over list/array indices

```
for i in range(len(somelist)):
    # Work with somelist[i]
```

Note:

- `range` returns a list of integers
- `range(a, b, s)` returns the integers `a, a+s, a+2*s, ...` up to *but not including* (!!) `b`
- `range(b)` implies `a=0` and `s=1`
- `range(len(somelist))` returns `[0, 1, 2]`

Let's replace our while loop by a for loop

```
v0 = 2
a = 0.2
dt = 0.1  # Increment
t_values = []
s_values = []
n = int(round(2/dt)) + 1  # No of t values
for i in range(n):
    t = i*dt
    s = v0*t + 0.5*a*t**2
    t_values.append(t)
    s_values.append(s)
print s_values  # Just take a look at a created list

# Make nicely formatted table
for t, s in zip(t_values, s_values):
    print '%.2f  %.4f' % (t, s)

# Alternative
for i in range(len(t_values)):
    print '%.2f  %.4f' % (t_values[i], s_values[i])
```

Traversal of multiple lists at the same time with `zip`

```
for e1, e2, e3, ... in zip(list1, list2, list3, ...):
```

Alternative: loop over a common index for the lists

```
for i in range(len(list1)):
    e1 = list1[i]
    e2 = list2[i]
    ...
```

Arrays

- List: collect a set of numbers or other objects in a single variable
- Lists are very flexible (can grow, can contain "anything")
- Array: computationally efficient and convenient list
- Arrays must have fixed length and can only contain numbers of the same type (integers, real numbers, complex numbers)
- Arrays require the `numpy` module

```
>>> import numpy
>>> L = [1, 4, 10.0]     # List of numbers
>>> a = numpy.array(L)   # Make corresponding array
>>> print a
[  1.   4.  10.]
>>> print a[1]
4.0
>>> print a.dtype        # Data type of an element
float64
>>> b = 2*a + 1
>>> print b
[  3.   9.  21.]
```

numpy functions creates entire arrays at once

```
>>> c = numpy.log(a)  # Take ln of all elements in a
>>> print c
[ 0.          1.38629436  2.30258509]
```

Create $n+1$ uniformly distributed coordinates in $[a, b]$:

```
t = numpy.linspace(a, b, n+1)
```

Let's use arrays in our previous program

```
import numpy
v0 = 2
a = 0.2
dt = 0.1  # Increment
n = int(round(2/dt)) + 1  # No of t values

t_values = numpy.linspace(0, 2, n+1)
s_values = v0*t + 0.5*a*t**2

# Make nicely formatted table
for t, s in zip(t_values, s_values):
    print '%.2f  %.4f' % (t, s)
```

Standard mathematical functions are found in the `math` module

```
>>> import math
>>> print math.sin(math.pi)
1.2246467991473532e-16          # Note: only approximate value
```

Get rid of the `math` prefix:

```
from math import sin, pi
print sin(pi)

# Or import everything from math
from math import *
print sin(pi), log(e), tanh(0.5)
```

Use the `numpy` module for standard mathematical functions applied to arrays

Matlab users can do

```
from numpy import *
```

The Python community likes

```
import numpy as np
print np.sin(np.pi)
```

Our convention: use `np` prefix, but not in formulas involving math functions

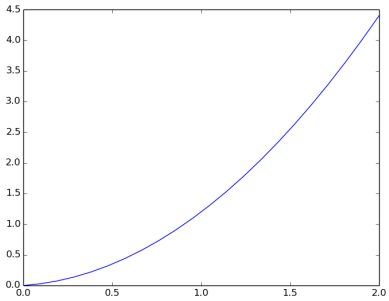
```
import numpy as np
from numpy import sin, exp
t = np.linspace(0, 4, 1001)
p = exp(-t)*sin(2*t)
```

Plotting

Plotting is done with `matplotlib`:

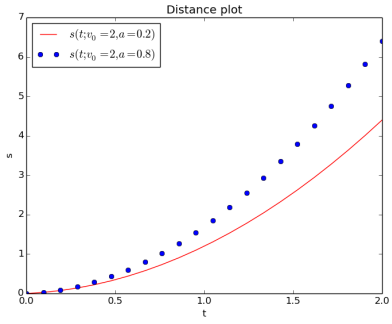
```
import numpy as np
import matplotlib.pyplot as plt

v0 = 0.2
a = 2
n = 21  # No of t values for plotting

t = np.linspace(0, 2, n+1)
s = v0*t + 0.5*a*t**2

plt.plot(t, s)
plt.savefig('myplot.png')
plt.show()
```

The plotfile `myplot.png` looks like

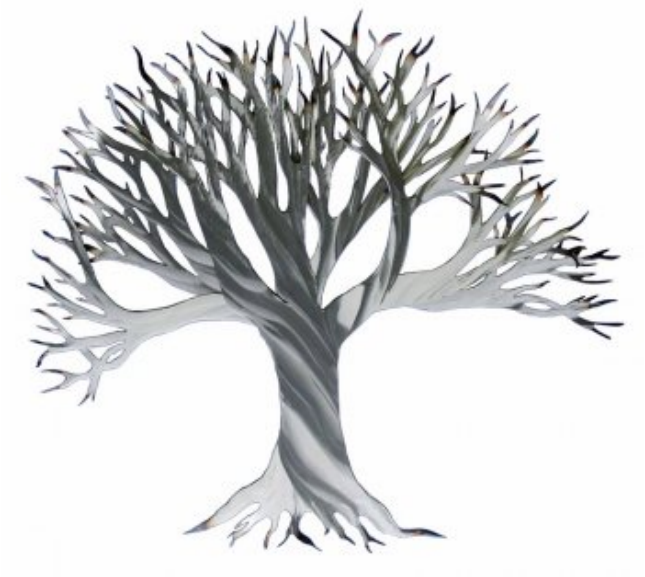


Plotting of multiple curves

```
import numpy as np
import matplotlib.pyplot as plt

v0 = 0.2
a = 2
n = 21  # No of t values for plotting

t = np.linspace(0, 2, n+1)
s = v0*t + 0.5*a*t**2

plt.plot(t, s)
plt.savefig('myplot.png')
plt.show()
```



Functions and branching



Functions

- $s(t) = v_0 t + \frac{1}{2} a t^2$ is a mathematical function
- Can implement $s(t)$ as a Python function `s(t)`

```
def s(t):
    return v0*t + 0.5*a*t**2

v0 = 0.2
a = 4
value = s(3)    # Call the function
```

Note:

- functions start with the keyword `def`
- statements belonging to the function must be indented
- function input is represented by arguments (separated by comma if more than one)
- function output is returned to the calling code
- `v0` and `a` are *global variables*
- `v0` and `a` must be initialzed before `s` is called

Functions can have multiple arguments

`v0` and `a` as function arguments instead of global variables:

```
def s(t, v0, a):
    return v0*t + 0.5*a*t**2

value = s(3, 0.2, 4)   # Call the function

# More readable call
value = s(t=3, v0=0.2, a=4)
```

Keyword arguments are arguments with default values

```
def s(t, v0=1, a=1):
    return v0*t + 0.5*a*t**2

value = s(3, 0.2, 4)          # specify new v0 and a
value = s(3)                  # rely on v0=1 and a=1
value = s(3, a=2)             # rely on v0=1
value = s(3, v0=2)            # rely on a=1
value = s(t=3, v0=2, a=2)     # specify everything
value = s(a=2, t=3, v0=2)     # any sequence allowed
```

- Arguments without the argument name are called *positional arguments*
- Positional arguments mustalways be listed before the keyword arguments in the function and in any call
- The sequence of the keyword arguments can be arbitrary

Vectorization speeds up the code

Scalar code (work with one number at a time):

```
def s(t, v0, a):
    return v0*t + 0.5*a*t**2

for i in range(len(t)):
    s_values[i] = s(t_values[i], v0, a)
```

Vectorized code: apply `s` to the entire array

```
s_values = s(t_values, v0, a)
```

How can this work?

- Array: `t`
- Expression: v0*t + 0.5*a*t**2
- `r1 = v0*t` (scalar times array)
- `r2 = t**2` (square each element)
- `r3 = 0.5*a*r2` (scalar times array)
- `r1 + r3` (add each element)

Python functions written for scalars normally work for arrays too!

True if computations involve arithmetic operations and math functions:

```
from math import exp, sin

def f(x):
    return 2*x + x**2*exp(-x)*sin(x)

v = f(4)  # f(x) works with scalar x

# Redefine exp and sin with their vectorized versions
from numpy import exp, sin, linspace
x = linspace(0, 4, 100001)
v = f(x)  # f(x) works with array x
```

However, if tests are not allowed:

```
def f(x):
    return -1 if x < 0 else x**4*exp(-x)*sin(x)

x = linspace(0, 4, 100001)
v = f(x)  # will not work
```

Python functions can return multiple values

Return $s(t) = v_0 t + \frac{1}{2}at^2$ and $s'(t) = v_0 + at$:

```
def movement(t, v0, a):
    s = v0*t + 0.5*a*t**2
    v = v0 + a*t
    return s, v

s_value, v_value = movement(t=0.2, v0=2, a=4)
```

`return s, v` means that we return a *tuple* ($\approx$ list):

```
>>> def f(x):
...     return x+1, x+2, x+3
...
>>> r = f(3)     # Store all three return values in one object r
>>> print r
(4, 5, 6)
>>> type(r)      # What type of object is r?
<type 'tuple'>
>>> print r[1]
5
```

Tuples are constant lists (cannot be changed)

Basic if-else tests

An if test has the structure

```
if condition:
    <statements when condition is True>
else:
    <statements when condition is False>
```

Here,

- `condition` is a boolean expression with value `True` or `False`.

```
if t <= t1:
    s = v0*t + 0.5*a0*t**2
else:
    s = v0*t + 0.5*a0*t1**2 + a0*t1*(t-t1)
```

Multi-branch if tests

```
if condition1:
    <statements when condition1 is True>
elif condition2:
    <statements when condition1 is False and condition2 is True>
elif condition3:
    <statements when condition1 and conditon 2 are False
     and condition3 is True>
else:
    <statements when condition1/2/3 all are False>
```

Just if, no else:

```
if condition:
    <statements when condition is True>
```

Implementation of a piecewisely defined function with if

A Python function implementing the mathematical function $s = v_0 t + \frac{1}{2}at^2$ reads

$$s(t) = \begin{cases} s_0 + v_0 t + \frac{1}{2}a_0 t^2, & t \leq t_1 \\ s_0 + v_0 t_1 + \frac{1}{2}a_0 t_1^2 + a_0 t_1 (t - t_1), & t > t_1 \end{cases}$$

```
def s_func(t, v0, a0, t1):
    if t <= t1:
        s = v0*t + 0.5*a0*t**2
    else:
        s = v0*t + 0.5*a0*t1**2 + a0*t1*(t-t1)
    return s
```

Python functions containing if will not accept array arguments

```
>>> def f(x): return x if x < 1 else 2*x
...
>>> import numpy as np
>>> x = np.linspace(0, 2, 5)
>>> f(x)
```

```
Traceback (most recent call last):
...
ValueError: The truth value of an array with more than one
element is ambiguous. Use a.any() or a.all()
```

Problem: `x < 1` evaluates to a boolean array, not just a boolean

Remedy 1: Call the function with scalar arguments

```
n = 201  # No of t values for plotting
t1 = 1.5

t = np.linspace(0, 2, n+1)
s = np.zeros(n+1)
for i in range(len(t)):
    s[i] = s_func(t=t[i], v0=0.2, a0=20, t1=t1)
```

Can now easily plot:

```
plt.plot(t, s, 'b-')
plt.plot([t1, t1], [0, s_func(t=t1, v0=0.2, a0=20, t1=t1)], 'r--')
plt.xlabel('t')
plt.ylabel('s')
plt.savefig('myplot.png')
plt.show()
```

Remedy 2: Vectorize the if test with `where`

Functions with if tests require a complete rewrite to work with arrays.

```
s = np.where(condition, s1, s2)
```

Explanation:

- `condition`: array of boolean values
- `s[i] = s1[i]` if `condition[i]` is `True`
- `s[i] = s2[i]` if `condition[i]` is `False`

Our example then becomes

```
s = np.where(t <= t1,
             v0*t + 0.5*a0*t**2,
             v0*t + 0.5*a0*t1**2 + a0*t1*(t-t1))
```

Note that `t <= t1` with array `t` and scalar `t1` results in a boolean array `b` where `b[i] = t[i] <= t1`.

Remedy 3: Vectorize the if test with array indexing

- Let `b` be a boolean array (e.g., `b = t <= t1`)
- `s[b]` selects all elements `s[i]` where `b[i]` is `True`
- Can assign some array expression `expr` of length `len(s[b])` to `s[b]`: `s[b] = (expr)[b]`

Our example can utilize this technique with `b` as `t <= t1` and `t > t1`:

```
s = np.zeros_like(t)  # Make s as zeros, same size & type as t
s[t <= t1] = (v0*t + 0.5*a0*t**2)[t <= t1]
s[t > t1]  = (v0*t + 0.5*a0*t1**2 + a0*t1*(t-t1))[t > t1]
```

Index