Intro to scientific Python programming

Hans Petter Langtangen[1,2]

Simula Research Laboratory[1]

University of Oslo[2]

Jan 8, 2015

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

Do you have access to Python?

Many methods:

- Mac and Windows: Anaconda
- Ubuntu: `sudo apt-get install`
- Web browser

See How to access Python for doing scientific computing for more details!

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 to 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 somelist:
    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`:

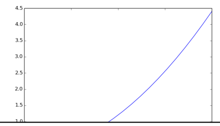
```
import numpy as np
import matplotlib.pyplot as plt

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



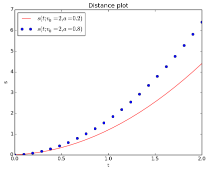
Plotting of multiple curves

```python
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

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

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

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

Note:

- `v0` and `a` are *global variables*
- `v0` and `a` must be initialzed before `s` is called

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

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

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