ENGG1003 - Monday Week 2

First steps: libraries & modules, plotting and printing

Steve Weller

University of Newcastle

February 27, 2021

Lecture overview

1. Python program with a library function §1.3
   - principles
   - live demo
2. importing from modules and packages §1.4
   - principles
   - live demo
3. simple plotting §1.5
   - principles
   - live demo
4. plotting and printing §1.6
   - principles
   - live demo

1) Python program with a library function

- describe the problem
- simple diagram: x, y, θ
- maybe a ball?
- algorithm is $\tan^{-1}$

The program

```
x = 10.0                  # Horizontal position
y = 10.0                  # Vertical position

angle = atan(y/x)

print((angle/pi)*180)
```

ball_angle_first_try.py

First use of a Python function

- first use of a *function*, in this case `atan`
- *argument*
- *return value*

Math review: radians and degrees

- Python's `atan` returns value in radians
- $\times \frac{180}{\pi}$ to get answer in degrees

Running the program

- screen grab from PyCharm – error message

Python standard library and import

- Python has plenty of functionality "built-in"
- LOTS more can be *imported*
- `atan` and other trigonometric functions not built in
- to activate that functionality, must explicitly import
- `atan` function is grouped together with many other mathematical functions in a *library module* called `math`

```
from math import atan, pi
```

The program: second attempt

```
from math import atan, pi

x = 10.0                # Horizontal position
y = 10.0                # Vertical position

angle = atan(y/x)

print((angle/pi)*180)
```

ball_angle.py

- script correctly produces 45.0 as output
- live demo in PyCharm shortly

Another way of importing

- use the import statement import math, but require `atan` and `pi` to be *prefixed* with math
- both techniques are commonly used and are the two basic ways of importing library code in Python

```
import math

x = 10.0                 # Horizontal position
y = 10.0                 # Vertical position

angle = math.atan(y/x)

print (angle/math.pi)*180
```

`ball_angle_prefix.py`

Live demo of Python program with a library function

2) Importing from modules and packages

motivation and context

(a) importing for use **without** prefix
(b) importing for use **with** prefix

Importing for use *without* prefix

```
from math import atan, pi

x = 10.0                # Horizontal position
y = 10.0                # Vertical position

angle = atan(y/x)

print((angle/pi)*180)
```

✓ Python code is easier to read

✗ allows name conflicts!

Name conflicts

- explain the basic idea
- do *not* explain example from text, which is too complicated
- will show an example shortly

Importing for use *with* prefix

```
import math

x = 10.0                # Horizontal position
y = 10.0                # Vertical position

angle = math.atan(y/x)

print (angle/math.pi)*180
```

✗ Python code is a little harder for humans to read

✓✓ eliminates name conflicts!

- **import with prefix is the standard and safer and preferred method of importing**

Avoiding name conflict using prefixes

```
import numpy
import math

x = numpy.exp([0, 1, 2])                    # do all 3 calculations
print(x)                                    # print all 3 results

y = math.cos(0)
print(y)
```

- `numpy` library includes an `exp` function
  - math review: exponential function $e^z = \texttt{exp(z)}$
- `math` library also includes an `exp` function—with a different implementation!

✓ **prefixes make clear which `exp` to use**

Imports with name change

```
import numpy as np
import math as m

x = np.exp([0, 1, 2])                  # do all 3 calculations
print(x)                               # print all 3 results

y = m.cos(0)
print(y)
```

- using as, `numpy` name becomes `np`
- similar for `math` and `m`

✓ Python code is easy to read

✓✓ eliminates name conflicts

Main modules used in ENGG1003

- `math`—description
- `numpy`—description
- `matplotlib`—description

Live demo of importing from modules and packages

3) Simple plotting

Context and problem setting

- xxx

Simple plot program

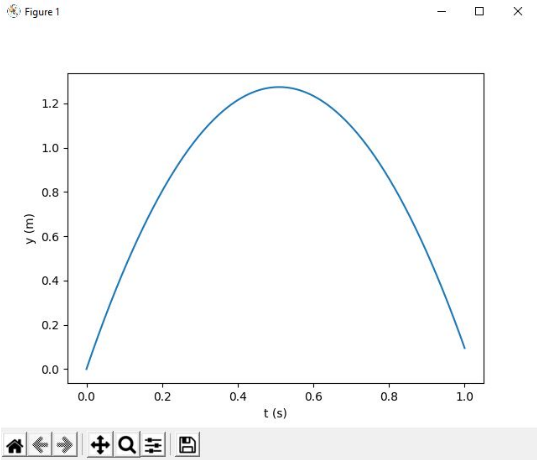
ball_plot.py

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

v0 = 5
g = 9.81
t = np.linspace(0, 1, 1001)

y = v0*t - 0.5*g*t**2

plt.plot(t, y)          # plots all y coordinates vs. all t coordinates
plt.xlabel('t (s)')     # places the text t (s) on x-axis
plt.ylabel('y (m)')     # places the text y (m) on y-axis
plt.show()              # displays the figure
```

- `linspace` function and our first *array*
- *vectorisation* in `y = v0*t - 0.5*g*t**2`
- plot commands

Program output

When we run `ball_plot.py` in PyCharm:



Our first array

```
t = np.linspace(0, 1, 1001)
```

- creates 1001 coordinates on the interval $[0, 1]$:
  $0, 0.001, 0.002, \ldots, 1$
- Python stores these as an *array*
- think of the array `t` as a collection of "boxes" in computer memory
- Python numbers these boxes consecutively from zero upwards:
  `t[0], t[1], t[2], ..., t[1000]`

Vectorization

```
y = v0*t - 0.5*g*t**2
```

- right-hand side is computed for every entry in the array `t`
- ie: for `t[0], t[1], t[2], ..., t[1000]`
- ✓ yields a collection of 1001 numbers in the result `y`, which (automatically) also becomes an array!
- technique of computing all numbers “in one chunk” is called *vectorization*

Plotting commands

Plotting commands are new, but simple:

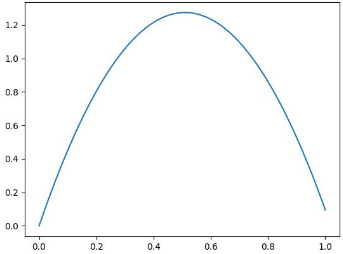
```
plt.plot(t, y)          # plots all y coordinates vs. all t coordinates
plt.xlabel('t (s)')     # places the text t (s) on x-axis
plt.ylabel('y (m)')     # places the text y (m) on y-axis
plt.show()              # displays the figure
```

Live demo of simple plotting

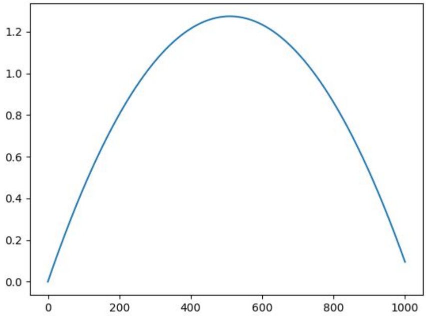
4) Plotting, printing and input data

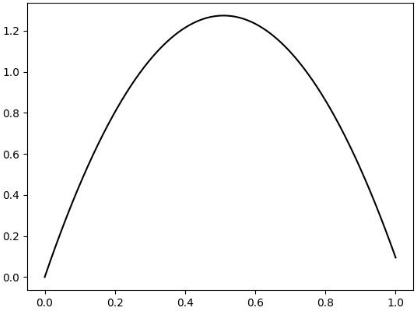
- `Matplotlib` is standard plotting package in Python
- have already seen array `y` (heights) plotted against another array `t` (points in time)

```
plt.plot(t,y)
```



Line styles

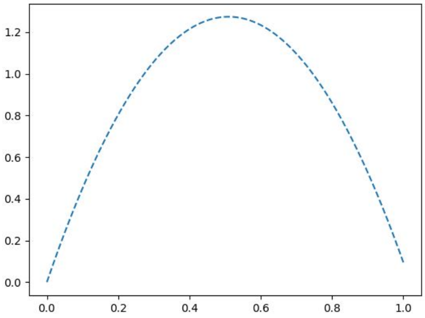


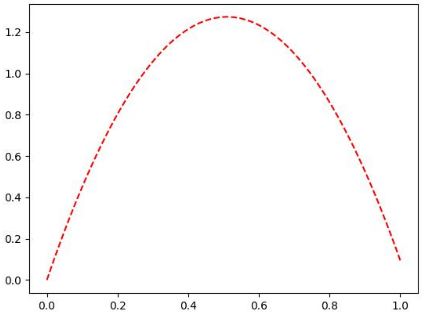


left: `plt.plot(y) # x-axis indices`

right: `plt.plot(t, y, 'k') # black line`

More line styles





left: `plt.plot(t, y, '--') # dashed`

right: `plt.plot(t, y, 'r--') # red dashed`

Plotting points only

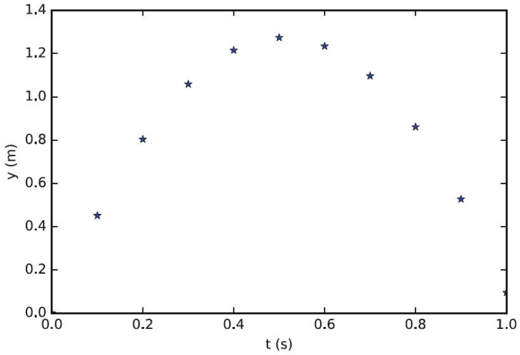
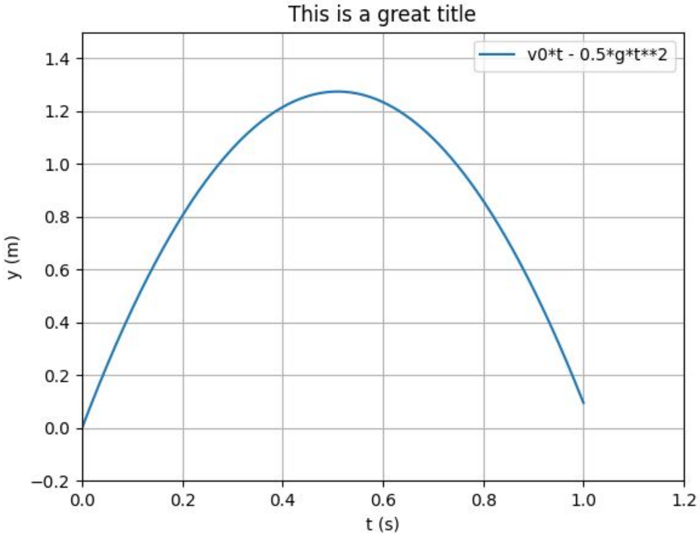


Fig. 1.2 Vertical position of the ball computed and plotted for every 0.1 s

```
t = np.linspace(0, 1, 11)   # 11 values give 10 intervals of 0.1
```

```
plt.plot(t, y, '*')     # default color, points marked with *
```

Decorating a plot



Decorating a plot

- add a legend

```
plt.legend(['v0*t - 0.5*g*t**2'])
```

- add a grid

```
plt.grid('on')
```

- display a title

```
plt.title('This is a great title')
```

- override default ranges for plot axes

```
plt.axis([0, 1.2, -0.2, 1.5])    # x in [0, 1.2] and y in [-0.2, 1.5]
```

Multiple curves in the same plot

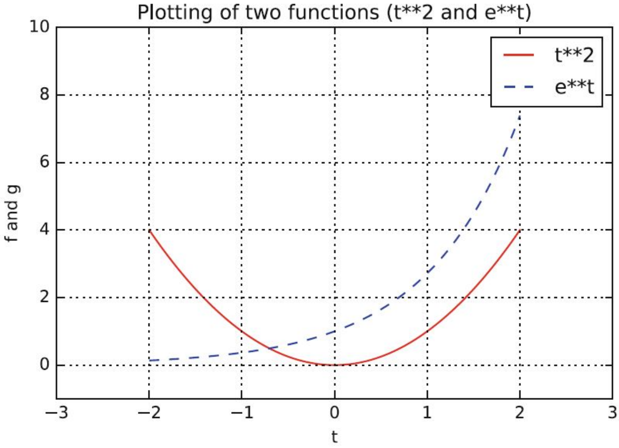


Fig. 1.3 The functions $f(t) = t^2$ and $g(t) = e^t$

Multiple curves in the same plot

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

```
t = np.linspace(-2, 2, 100)   # choose 100 points in time interval

f_values = t**2
g_values = np.exp(t)

plt.plot(t, f_values, 'r', t, g_values, 'b--')
plt.xlabel('t')
plt.ylabel('f and g')
plt.legend(['t**2', 'e**t'])
plt.title('Plotting of two functions (t**2 and e**t)')
plt.grid('on')
plt.axis([-3, 3, -1, 10])
plt.show()
```

Key line of code for multiple curves:

```
plt.plot(t, f_values, 'r', t, g_values, 'b--')
```

Printing variables and strings

- print the value of variable `y`

```
print(y)
```

- print the string `This is some text`

```
print('This is some text')
```

  - enclose text in single quotes

Print *one* variable and text combined

- if variable `v1` has value 10.0 and you want to display as follows:

```
v1 is 10.0
```

- use the following Python code:

```
print('v1 is {}'.format(v1))
```

  - pair of curly brackets {} acts as a *placeholder*
  - ...says *where* to place value
  - `.format(v1)` converts variable `v1` to string `10.0`

Printing *several* variables and text combined

- if `v1` and `v2` have values 10.0 and 20.0 and you want to display as follows:

```
v1 is 10.0, v2 is 20.0
```

- use the following Python code:

```python
print('v1 is {}, v2 is {}'.format(v1, v2))
```

  - now *two* pairs of curly brackets {} act as a placeholders
  - `.format(v1,v2)` dictates the order in which placeholders are filled
  - in this case: `v1` first, then `v2`

Controlled printing: decimals, scientific notation & strings

- real number 12.89643
- integer 42
- string `some message`

- supose you want to display as follows:

```
real=12.896, integer=42, string=some message
real=1.290e+01, integer=   42, string=some message
```

- use the following Python code:

```
r = 12.89643            # real number
i = 42                  # integer
s = 'some message'      # string    (equivalent: s = "some message")

print('real={:.3f}, integer={:d}, string={:s}'.format(r, i, s))
print('real={:9.3e}, integer={:5d}, string={:s}'.format(r, i, s))
```

Controlled printing: decimals, scientific notation & strings

- first call to `print`

```
print('real={:.3f}, integer={:d}, string={:s}'.format(r, i, s))
```

  - `:.3f` write number `r` compactly using 3 decimals
  - `:d` write integer `i` as compactly as possible
  - `:s` write string `s`

- second call to `print`

```
print('real={:9.3e}, integer={:5d}, string={:s}'.format(r, i, s))
```

  - `:9.3e` write number `r` in *scientific notation* with 3 decimals in field of width 9 characters
  - `:5d` write integer `i` in field of width 5 characters

Live demo of plotting and printing