

ENGG1003 - Monday Week 2

First steps: importing from modules, plotting and printing

Steve Weller

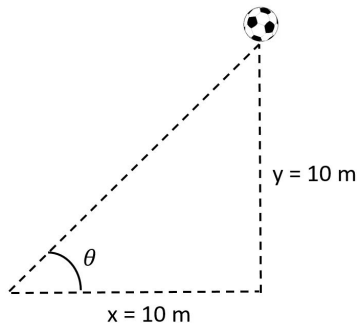
University of Newcastle

February 28, 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



- **Aim:** calculate angle θ
- using trigonometry,
 $\tan(\theta) = y/x$
- **Algorithm:** $\theta = \tan^{-1}(y/x)$
- Math review: $\tan^{-1}(z)$
calculates the angle θ such
that $\tan(\theta) = z$

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

```
angle = atan(y/x)
```

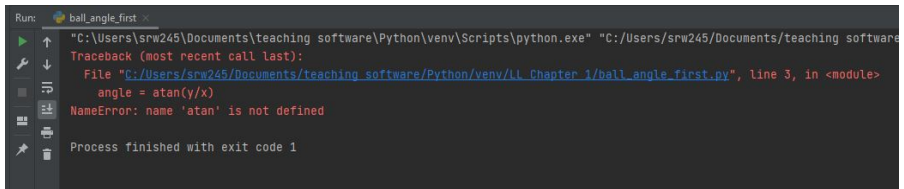
- our first use of a *function*, in this case `atan`
 - ▶ corresponds to \tan^{-1} in mathematics
- line of code above shows how function `atan` is *called*
- ratio y/x is the *argument* of function `atan`
- computed value is *returned* from `atan`
 - ▶ ...and result is assigned to `angle`

Math review: radians and degrees

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

- Python's `atan` function returns angle in *radians*
- multiply radians by $180/\pi$ to convert to degrees

Running the program in PyCharm



The screenshot shows the Run console in PyCharm. The title bar indicates the file is 'ball_angle_first'. The console output shows the execution of a Python script using the interpreter 'C:\Users\srw245\Documents\teaching software\Python\venv\Scripts\python.exe'. A traceback is displayed, indicating a 'NameError: name 'atan' is not defined' at line 3 of the file 'C:\Users\srw245\Documents\teaching software\Python\venv\ll Chapter 1\ball_angle_first.py'. The code snippet shown is 'angle = atan(y/x)'. The console also shows 'Process finished with exit code 1'.

```
Run: ball_angle_first x
"C:\Users\srw245\Documents\teaching software\Python\venv\Scripts\python.exe" "C:/Users/srw245/Documents/teaching software
Traceback (most recent call last):
  File "C:/Users/srw245/Documents/teaching software/Python/venv/ll Chapter 1/ball_angle_first.py", line 3, in <module>
    angle = atan(y/x)
NameError: name 'atan' is not defined
Process finished with exit code 1
```

```
angle = atan(y/x) \\  
NameError: name 'atan' is not defined
```

- **Problem:** Python does not have `atan` function “built-in”!

Python standard library and import

- Python has some functionality built-in
- ...but 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

- Python has many libraries
 - importing what's needed (and only what's needed) is sensible
- (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 $e^z = \exp(z)$
 - `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 (“nickname”)
- similar for math and m
- ✓ Python code is easy to read
- ✓✓ eliminates name conflicts

Main libraries used in ENGG1003

- **math**—basic mathematical operations & constants
 - ▶ math functions defined in C programming language
- **numpy**—numerical Python
 - ▶ large collection of powerful mathematical functions for scientific computing
 - ▶ handles large datasets, matrices and arrays
- **matplotlib**—visualization
 - ▶ comprehensive library for creating static, animated, and interactive visualizations
 - ▶ syntax closely follows MATLAB programming language

Live demo of importing

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)

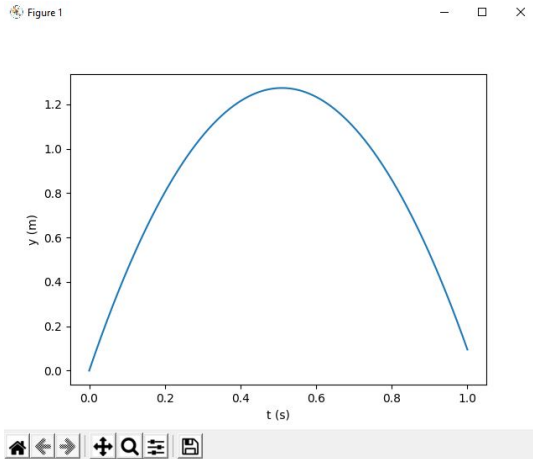
y = v0*t - 0.5*g*t**2

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 $[0, 1]$:
 $0, 0.001, 0.002, \dots, 1$
- 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:

$t[0], t[1], t[2], \dots, t[1000]$

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:

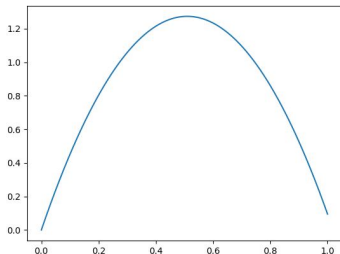
```
plt.plot(t, y)      # plots all y coordinates vs. all t coordinates
plt.xlabel('t (s)')  # places the text t (s) on x-axis
plt.ylabel('y (m)')  # places the text y (m) on y-axis
plt.show()          # displays the figure
```

Live demo of simple plotting

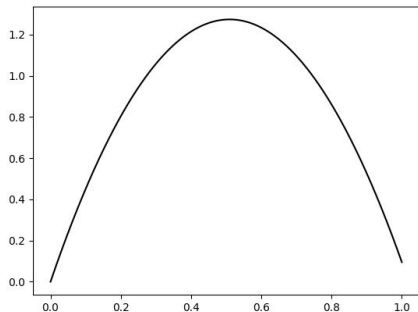
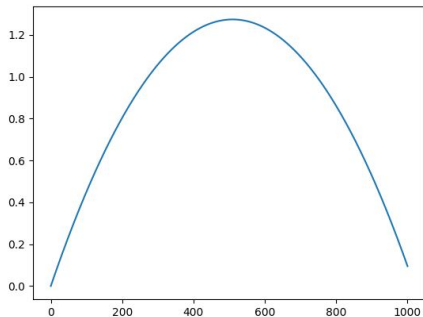
4) Plotting, printing and input data

- 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)`



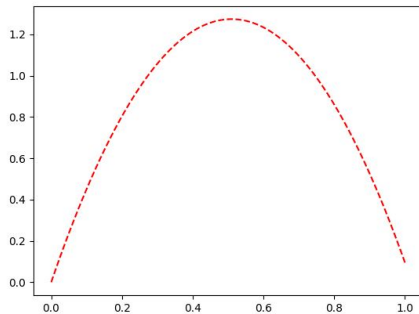
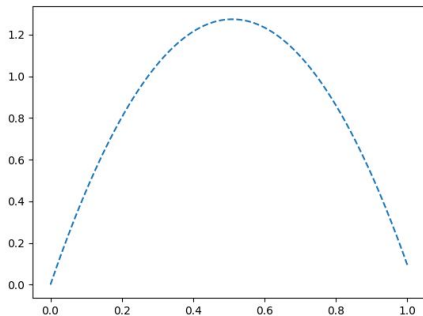
Line styles



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

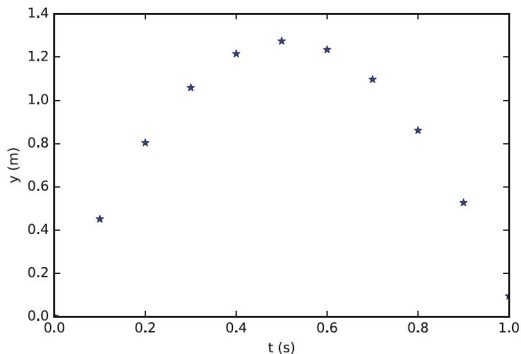
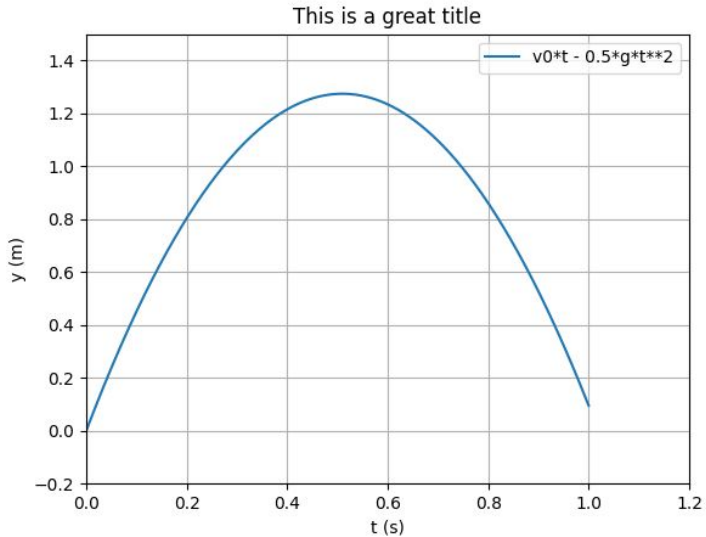


Fig. 1.2 Vertical position of the ball computed and plotted for every 0.1 s

```
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

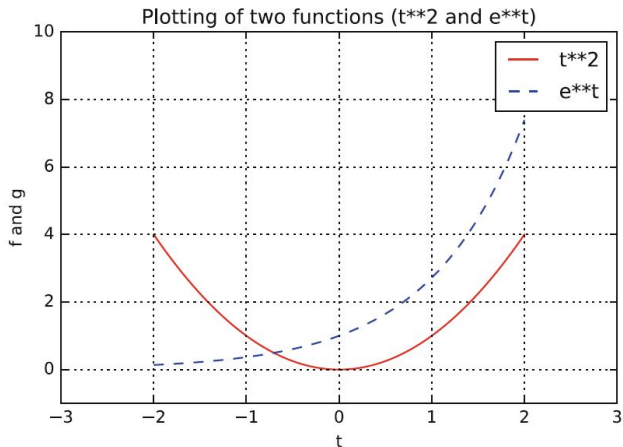


Fig. 1.3 The functions $f(t) = t^2$ and $g(t) = e^t$

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:

```
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
- suppose 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