pythonIntro

August 8, 2025

[1]:
```
"""
In each chapter of these notes, all imported modules and libraries
will be listed at the top of the document. Specialized imports will
be found within this document to better explain the context in which
they are used.
"""
import math # mathematical functions
import numpy # a numerical methods library
import matplotlib.pyplot as plt # basic plotting functions
```

1 Chapter 1: Introduction to Python

Q: Why are we going to use Python instead of MATLAB for our scientific computing work?

A:

1) Python currently is the most-used programing language in the world so programming in Python is a much more useful skill for students
2) Python has just as strong of scientific computing libraries as MATLAB
3) Same for plotting
4) Python has true object-oriented capabilities, as opposed to MATLAB's file-based approach
5) Python works very well in interfacing to real-time control systems and the web
6) Python can be run in a Jupyter notebook, making possible reports with live code and text
7) Python is free!

1.1 Getting started

Step 0: Download a **free** Python 3 distribution, such as from anaconda.com/download for Mac or PC

Step 1: Get started via one of three ways

- Open a terminal or console window and at the prompt, type "python" to start the Python interactive command shell where Python statements typed in after the prompt "»>" will immediately be interpreted. For example, "»> 2+3" will return "5"; use the **built-in** Python function "quit()" to end the interactive Python session.

- Or by typing into a terminal or console window "jupiter notebook" to start a new notebook or open an existing notebook, such as this one, with Markdown (typeset LaTeX comments) and Code cells (Python and other programming languages); to switch a cell to Code or Markdown, use the right-most tab of the second top menu bar. New cells are added with the "+" icon, cut with the scissors icon, and moved using the "Edit" tab. Note that to typeset the markdown windows or execute the code cells, click on the "run" icon ▶ or use one of the options under the "Run tab." Note that a convenient way to save the output of a Jupyter Notebook is to use the "File > Save and Export Notebook As... > PDF" Of course, the resulting pdf will not have the Notebook functionality, but it will be readable to anyone with a pdf reader.
- Or by typing into a terminal or console window "spyder" to start the Anaconda Spyder Interactive Development Environment (IDE) that combines text editing for Python code, windows for displaying program test output and plots, as well as debugging tools. Note that Jupyter Notebook and Spyder can be opened using the Anaconda Navigator app.

Python files are plain text with filenames of the form "filename.py" and can be edited by any standard text editor including TextEdit on a Mac and Notepad on a PC (in both cases, be sure to save the file **in plain text format**). Jupyter Notebooks, opened, edited, and run through a browser (e.g., Chrome) interface, are not human readable, and have filenames of the form "filename.ipynb" The easiest way to run a Python script is to open a Terminal (Mac) or Consol/Terminal (PC) window, navigate to the directory containing the file to be run (using the Unix "cd" command on a Mac or PC – "pwd" returns the present working directory on both a Mac and PC, 'ls' lists the files/folders in the current directory), and then running the script by typing "python filename.py; for a Jupyter Notebook file,"jupyter notebook filename.ipynb"

1.2 Python variables and arithmatic operations

With the interactive command shell, we can use Python as a calculator:

[2]:
```
5*7-9
```

[2]:
```
26
```

[3]:
```
5*(7-9) # note that the standard rules of precedence apply
```

[3]:
```
-10
```

Note that any text after the "#" is considered a comment

[4]:
```
2**4
```

[4]:
```
16
```

[5]:
```
4**-2
```

[5]:
```
0.0625
```

[6]:
```
abs(-11)
```

[6]:
```
11
```

[7]:
```
int(4/3)
```

[7]:
```
1
```

[8]:
```
a,b = -3,5 # make two assignments in one operation
print(a,b)
```

```
-3 5
```

[9]:
```
a,b = b,a # exchange values
print(a,b)
```

```
5 -3
```

[10]:
```
x = 1e-6 # standard scientific notation
print(x)
```

```
1e-06
```

Formatted printing - consider

[11]:
```
x = 1/3
print(x)
```

```
0.3333333333333333
```

Skip a line:

[12]:
```
print('One third:\n',x)
```

```
One third:
 0.3333333333333333
```

[13]:
```
print('1/3 = {:.2f}, 2/3 = {:.3f}, and 4/3 = {:.2e}' \
      .format(x,2*x,4*x))
```

```
1/3 = 0.33, 2/3 = 0.667, and 4/3 = 1.33e+00
```

There is a lot going on here with the format specification {:.2f}, line-continuation marker \, and .function() notation - more on all of these later.

Final notes on variables We note that Python is **case sensitive**

[14]:
```
b = 1
B = 2
b == B # more on the definition of equivalence == later
```

[14]:
```
False
```

Importing modules What about other common mathematical functions, such as cos, sin, exp, pi, or sqrt? $\Longrightarrow$ the *math* module must be **imported**:

[15]:
```
import math # just for illustation - we've already done this
math.cos(math.pi)
```

[15]:
```
-1.0
```

[16]:
```
math.sqrt(9)
```

[16]:
```
3.0
```

Test the following to find more information on the math module:

[17]:
```
# dir(math)
# help(math)
# help(math.cos)
```

Note that some math functions are **built-in**

[18]:
```
print( int(math.pi) )
print( max( [1,2,5,-2,0] ) )
print( len( 'a string' ) )
```

```
3
5
8
```

For more information, type "»>help(math)" – note that "math" must have been imported.

Q: How do we obtain a list of built-in functions?

1.3 Strings and lists

A **string** is a segment of text:

[19]:
```
M = 'The moon orbits the Earth'
print(M)
print(M[0:5])
len(M)
print(2*M)
```

```
The moon orbits the Earth
The m
The moon orbits the EarthThe moon orbits the Earth
```

[20]:
```
a,b,c = 'cat',' ','dog'
a+b+c # string concatenation
```

[20]:
```
'cat dog'
```

[21]:
```
a + str(2) + c
```

[21]:
```
'cat2dog'
```

[22]:
```
# try M[4] = 'Z' as a demonstration that strings are immutable
```

A **list** is an indexed array (starting with index 0) that can have elements of all the same type:

[23]:
```
y = [1, 2, 3, 4, 5, 6]
print(y)
print('length:',len(y))
print(y[0],y[1])
print(y[-2],y[-1])
```

```
[1, 2, 3, 4, 5, 6]
length: 6
1 2
5 6
```

or of different types:

[24]:
```
w = [1, 'cat', [2, 3]]
print(w)
```

```
[1, 'cat', [2, 3]]
```

A list is *mutable*:

[25]:
```
w[1] = 'dog'
print(w)
w[2][0] = 10
print(w)
```

```
[1, 'dog', [2, 3]]
[1, 'dog', [10, 3]]
```

and can change in length:

[26]:
```
w.append('fish')
print(w)
```

```
[1, 'dog', [10, 3], 'fish']
```

[27]:
```
w.pop(2) # note () and not []
print(w)
```

```
[1, 'dog', 'fish']
```

but be careful...

[28]:
```
z = y
z[2] = 'cat'
print('Modified "copy" of y:',z)
```

```
Modified "copy" of y: [1, 2, 'cat', 4, 5, 6]
```

[29]:
```
print('"Original" y:',y)
```

```
"Original" y: [1, 2, 'cat', 4, 5, 6]
```

Lists of integers can be generated as **range** objects

[30]:
```
Q = range(5)
print(Q) # note how this is a range object and not a list
for q in Q:
    print(q)
```

```
range(0, 5)
0
1
2
3
4
```

note that in this case the range starts at 0 and contains 5 elements.

List slices using the : operator We can extract a portion of a list using the ":" operator

[31]:
```
q = ['A','B','C','D','E','F','G','H']
print(q[:]) # same as print(q)
print(q[1:4]) # print q[1], q[2], q[3] but not q[4]
print(q[4:])
print(q[1:-4]) # does not print q[-4]
```

```
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
['B', 'C', 'D']
['E', 'F', 'G', 'H']
['B', 'C', 'D']
```

To avoid confusion when working with n-dimensional arrays, it's best to use a numpy.array()

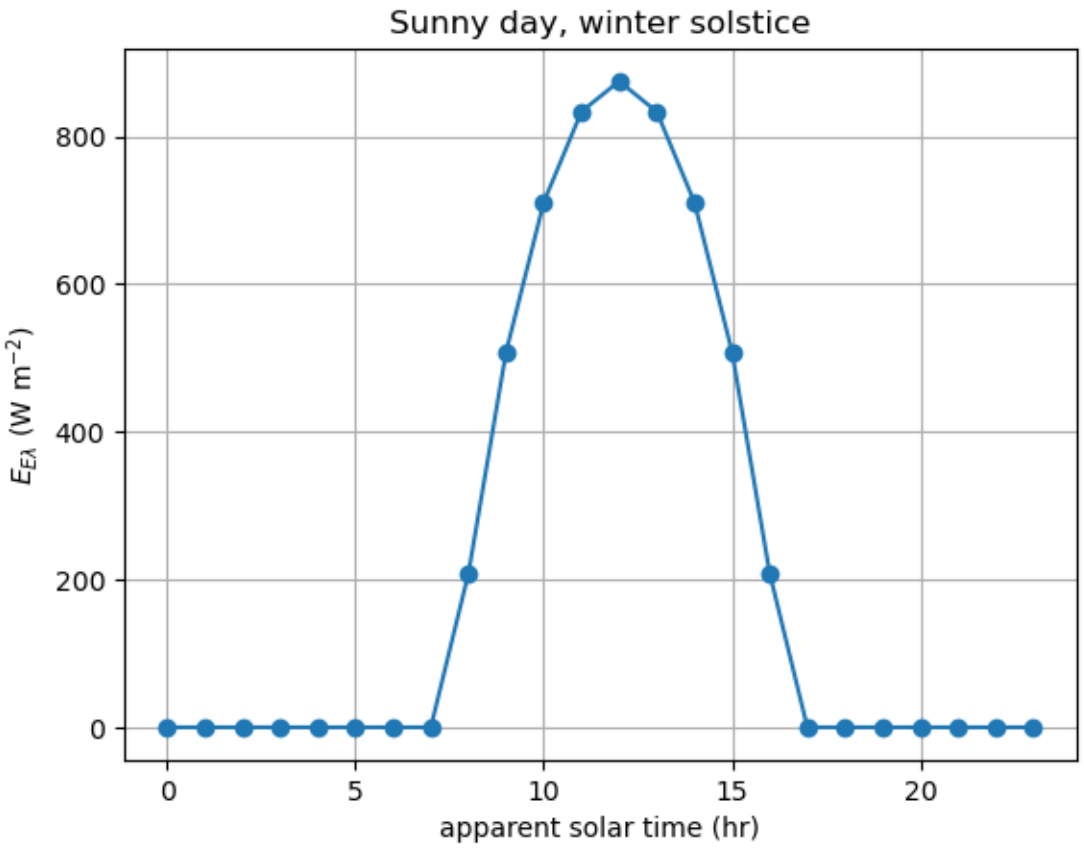
[32]:
```
import numpy # again, already done - just for illustration
q = numpy.array([ ['A','B','C'], ['D','E','F'] ])
print('q:\n',q)
print('a column of q:',q[:,1])
print('a row of q:',q[1,:])
```

```
q:
 [['A' 'B' 'C']
 ['D' 'E' 'F']]
a column of q: ['B' 'E']
a row of q: ['D' 'E' 'F']
```

1.3.1 A look ahead to plotting

Now that we have a basic understanding of lists and have introduced the concept of importing Python modules, we can jump ahead to the practical issue of plotting data. Consider, for example, the solar energy analysis of a stand-alone bicycle path illumination system located in Takoma Park, MD. The individual PV modules are tilted 50^o N and during the winter solstace, we can plot the solar irradiance (incident total solar power) by the following:

[33]:
```
import matplotlib.pyplot as plt

ast = [0,1,2,3,4,5,6,7,8,9,10,11,12, \
       13,14,15,16,17,18,19,20,21,22,23]
EG = [0,0,0,0,0,0,0,0,208.5,508.3,710.9,833.4, \
      874.6,833.4,710.9,508.3,208.5,0,0,0,0,0,0,0]

plt.plot(ast,EG,'o-')
plt.xlabel('apparent solar time (hr)')
plt.ylabel('$E_{E\lambda}$ (W m$^{-2}$)')
plt.title('Sunny day, winter solstice')
plt.grid() # note lack of input parameters, but still need ()
plt.show() # plot object is created before it is displayed
```



Note that 1) the ordinate has units W m^{-2} where the superscript "2" is typset using LaTeX syntax 2) we can also write this *script* using a standard text editor, saving it as "myPlot.py," and then running it by typing "python myPlot.py"

1.4 Dictionaries, tuples, and sets

A **dictionary** is an effective way to represent data in (ordered, post Python 3.7) key:value form:

[34]:
```
D = { 'Name':'Ray', 'Office':2147, 'Degrees':['BS','PhD'] }
Cp = { 'air':29.19, 'copper':24.47, 'methanol':68.62, 'water':75.38, 'units':'J/
 ↪(mol K)' }
print(Cp)
print(len(Cp))
```

```
{'air': 29.19, 'copper': 24.47, 'methanol': 68.62, 'water': 75.38, 'units':
'J/(mol K)'}
5
```

[35]:
```
key = 'methanol'
value = Cp[key]
print(key,value,Cp['units'])
```

```
methanol 68.62 J/(mol K)
```

Tuples are similar to lists and so they are ordered, but unlike sets they are immutable; the most frequent use of tuples is in the returned values of a **function** (to be discussed later):

[36]:
```
T = ('yellow', 'green', 'red') # a tuple
print('from tuple definition:',T[1])

def myFun():
    a,b,c = 'yellow','green','red'
    return a,b,c
X = myFun() # X will be a tuple
print('from function returned values:',X[1])
```

```
from tuple definition: green
from function returned values: green
```

Sets are unordered, allow for elements to be removed and added, and support a number of traditional (built-in) set operations:

[37]:
```
S = {'Ar', 'B', 'C', 'Ar'} # note duplicate element 'A'
T = {'B', 'Fe', 'Zn', 'Ar'}
print('orignal set S:',S)
S.add('F')
S.remove('C')
print('modified set S:',S)
```

```
orignal set S: {'B', 'Ar', 'C'}
modified set S: {'B', 'F', 'Ar'}
```

[38]:
```
print('union:',S|T)
print('intersection:',S&T)
```

```
union: {'B', 'F', 'Ar', 'Zn', 'Fe'}
intersection: {'B', 'Ar'}
```

Q: What are the four built-in Python data types?

1.5 Logical operations

The key concept is to distinguish between *assignment* operations:

[39]:
```
x = 2; print(x) # note use of ";" to write 2 statements on one line
```

```
2
```

and checking for *equivalence*:

[40]:
```
x == 2
```

[40]: True

[41]:
```
x == 4
```

[41]: False

[42]:
```
x != 4
```

[42]: True

[43]:
```
x < 2
```

[43]: False

[44]:
```
x <= 2
```

[44]: True

Note that Boolean values also can be 0 and 1:

[45]:
```
True == 0
```

[45]: False

[46]:
```
True == 1
```

[46]: True

[47]:
```
False == 0
```

[47]: True

[48]:
```
False == 1
```

[48]: False

and that the result of a logical expression can be assigned to a variable

[49]:
```
result = 2 == 3
print(result)
```

```
False
```

Compound logical expressions Consider using the "and" and "or" logical operations

[50]:
```
a,b = 2,-1
a < 5 and b < 5
```

[50]: True

[51]:
```
a < -2 or b > -2
```

[51]: True

Precedence of operations Earlier in the lecture notes we observed in the operation 5*7-9 that multiplication takes place first followed by subtraction. Logical operations also follow the rules of precedence with operations in which the equivalence operations are carried out first, followed by the "or" evaluation in the following example

[52]:
```
2 == 3 or 4 == 4
```

[52]: True

We can summarize the order of precedence as:

1. operations in parentheses () are evaluated first (this applies to functions)
2. exponentiation ** and modulo operations
3. then unitary minus (-)
4. * and /
5. + and -
6. the logical operators <, >, <=, >=, ==, !=
7. logical and
8. logical or
9. assignment = is performed last

When two operations with equal precedence are encountered, Python evaluates them from left to right. Consider, for example, a statement that at first seems impossible

[53]:
```
y = 2**4/2 and True + 1
print(y)
```

2

Q: Why does the statement "4**-2" produce the correct result?

1.5.1 White space

This leads to the definition of the "if/else" structure, taking note of the four (white) space indentation which indicates that the indented statement is to be executed if the logical expression evaluates to "True"

[54]:
```
x = 3
if x == 2:
```

```
    print('x is two')
else:
    print('x is not two')
print('all done')
```

```
x is not two
all done
```

Let's now consider writing a script to calculate the *real* roots of the quadratic equation

$$y = ax^2 + bx + c$$

the roots of which are found as

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \qquad x = \frac{-b \pm \sqrt{\Delta}}{2a} \qquad \text{with } \Delta = b^2 - 4ac$$

[55]:
```
a,b,c = 1,3,2
Delta = b**2 - 4*a*c
if Delta >= 0:
    x0,x1 = (-b + math.sqrt(Delta))/(2*a),(-b - math.sqrt(Delta))/(2*a)
    print('roots x0 = {:.2f}, x1 = {:.2f}'.format(x0,x1))
else:
    print('no real roots')
```

```
roots x0 = -1.00, x1 = -2.00
```

1.6 Iterative structures

Frequently, we make use of statements that incrementally change the value of a variable:

[56]:
```
n = 10
n = n+1
print(n)
```

```
11
```

that in Python can be written more compactly as

[57]:
```
n += 1
print(n)
```

```
12
```

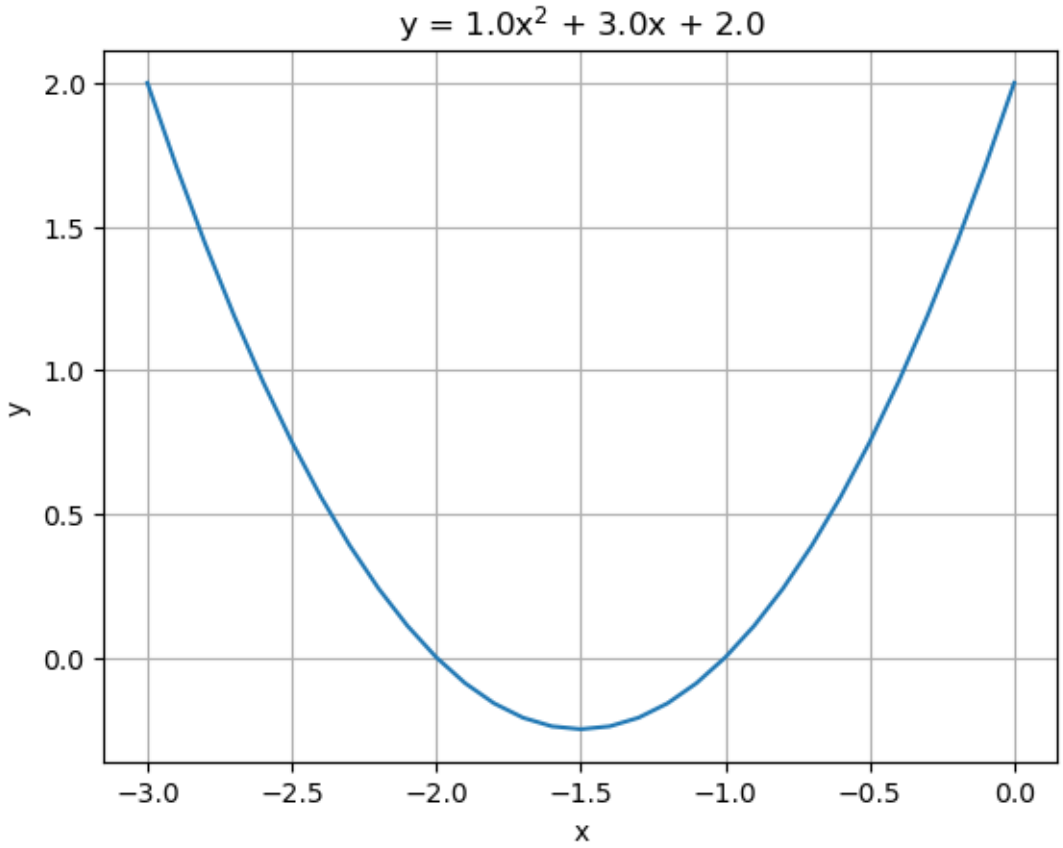
Note that this condensed notation works with other basic mathematical operations, e.g.

[58]:
```
n /= 2
print(n)
```

```
6.0
```

We can use this to create a plot of our quadratic equation $y = ax^2 + bx + c$

[59]:
```
a,b,c = 1,3,2
x = 0
y = a*x**2 + b*x + c
xPlot,yPlot = [x],[y] # note how blank lines below are ignored

while x >= -3:
    x -= 0.1
    y = a*x**2 + b*x + c
    xPlot.append(x); yPlot.append(y)

plt.plot(xPlot,yPlot)
plt.xlabel('x'); plt.ylabel('y')
plt.title('y = {:.1f}x$^2$ + {:.1f}x + {:.1f}'.format(a,b,c))
plt.grid()
plt.show()
```



Integer sequences also can be created using the Python *built-in* **range**(start,stop,step) function

where start=0 and step=1 are defaults:

[60]:
```
X = range(5)
for x in X: # x != X case sensitive!
    print(x)
```

```
0
1
2
3
4
```

Note the use of the for-in structure to iterate through the elements of the range object. Likewise,

[61]:
```
for x in range(-2,2,1):
    print(x)
```

```
-2
-1
0
1
```

For cases when floating-point sequences are needed, such as the x-axis of our quadratic equation ploting script, we can use the **numpy.linspace**(start,stop,num) function

[62]:
```
x = numpy.linspace(-3,0,31)
print(x)
print(type(x))
```

```
[-3.  -2.9 -2.8 -2.7 -2.6 -2.5 -2.4 -2.3 -2.2 -2.1 -2.  -1.9 -1.8 -1.7
 -1.6 -1.5 -1.4 -1.3 -1.2 -1.1 -1.  -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3
 -0.2 -0.1  0. ]
<class 'numpy.ndarray'>
```

Note that x actually is a numpy.ndarray and so operations such as x**2 will work on an element-by-element basis, whereas this operation would not work if x was a list.

1.6.1 List comprehensions

The same sequence can be created using a **list comprehension**:

[63]:
```
X = [ x/10 - 3 for x in range(31) ]
print(X)
```

```
[-3.0, -2.9, -2.8, -2.7, -2.6, -2.5, -2.4, -2.3, -2.2, -2.1, -2.0, -1.9, -1.8,
-1.7, -1.6, -1.5, -1.4, -1.3, -1.2, -1.1, -1.0, -0.8999999999999999,
-0.7999999999999998, -0.7000000000000002, -0.6000000000000001, -0.5,
-0.3999999999999999, -0.2999999999999998, -0.20000000000000018,
-0.10000000000000009, 0.0]
```

Q: Why are there elements such as -0.7000000000000002 instead of -0.7?

A Consider the concept of **machine epsilon**:

[64]:
```
x = 1
for i in range(100):
    x /= 10
    stop = x+1 == 1
    print(i,x,stop)
    if stop:
        break
```

```
0 0.1 False
1 0.01 False
2 0.001 False
3 0.0001 False
4 1e-05 False
5 1.0000000000000002e-06 False
6 1.0000000000000002e-07 False
7 1.0000000000000002e-08 False
8 1.0000000000000003e-09 False
9 1.0000000000000003e-10 False
10 1.0000000000000003e-11 False
11 1.0000000000000002e-12 False
12 1.0000000000000002e-13 False
13 1.0000000000000002e-14 False
14 1e-15 False
15 1.0000000000000001e-16 True
```

1.6.2 Quadrature

Returning to our solar irradiance data, E_G (in W m^{-2}) as a function of *ast* (h after solar midnight), we can convert the instantanious **power** over the course of the day to the total solar **energy** collected over the day through **quadrature** (numerical integration) using the **trapezoidal rule**, with the calculated energy expressed in terms of the conventional solar industry unit kWh m^-2.

[65]:
```
print(ast[0:12],EG[0:12]) # check to make sure the lists are still defined
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] [0, 0, 0, 0, 0, 0, 0, 0, 208.5, 508.3,
710.9, 833.4]
```

[66]:
```
totEnergy = 0
for i in range(len(ast)-1):
    totEnergy += (EG[i+1] + EG[i])/2*(ast[i+1]-ast[i])/1000 # in kWh m^-2
print('total energy: {:.1f} kWh m^-2'.format(totEnergy))
```

```
total energy: 5.4 kWh m^-2
```

Q: What are the implications of this quantity relative to the storage capacity of a 100 kWh EV?

1.6.3 Finite-difference estimates of derivatives

If we have two lists of data, e.g., x and y, with each list of length N, we can approximate the derivatives of these discretized curves using **forward finite differences**

$$\frac{dy}{dx} \approx \frac{y_{n+1} - y_n}{x_{n+1} - x_n} \quad n = 0, 1, N-2$$

For our quadratic equation $y = ax^2 + bx + c$, the exact 1st derivative is

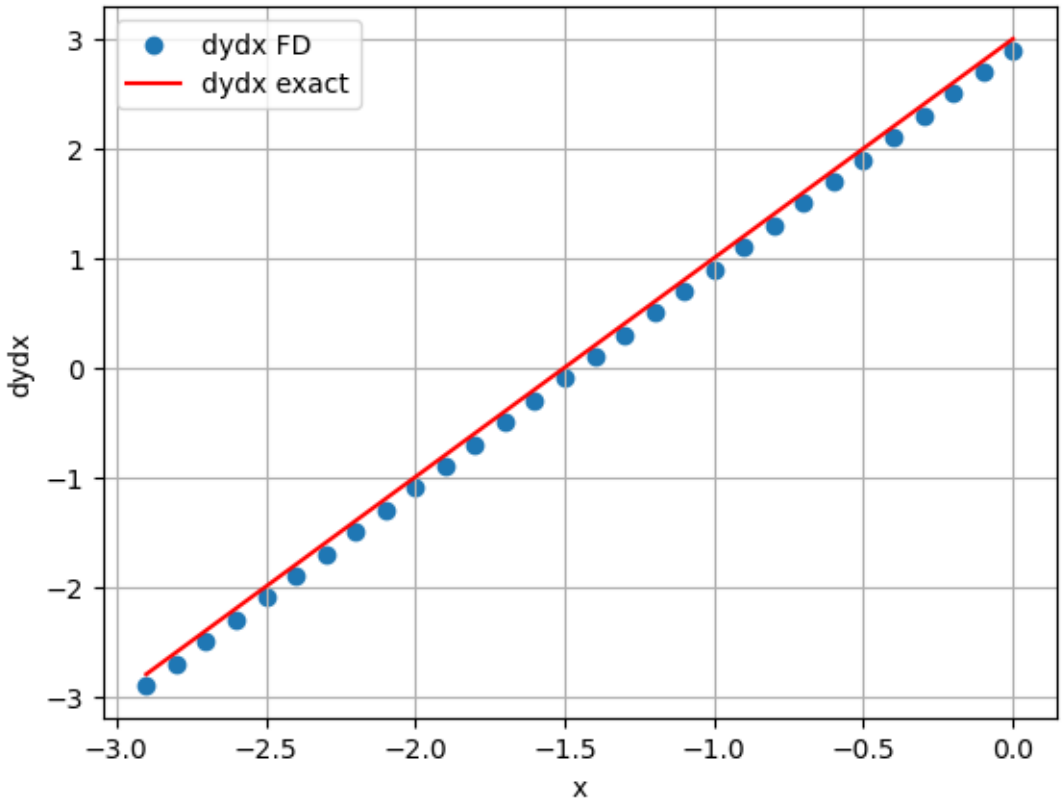
$$\frac{dy}{dx} = 2ax - b$$

and so we compare our exact and numerical derivatives as:

[67]:
```
print(xPlot[0:6],yPlot[0:6]) # check if our xPlot, yPlot values are still stored
print('a,b:',a,b) # also to check
```

```
[0, -0.1, -0.2, -0.30000000000000004, -0.4, -0.5] [2, 1.71, 1.44, 1.19, 0.96,
0.75]
a,b: 1 3
```

[68]:
```
N = len(xPlot) # assume yPlot is the same length
dydxFD = []
dydxExact = []
for n in range(N-1):
    dydxFD.append( (yPlot[n+1]-yPlot[n])/(xPlot[n+1]-xPlot[n]) )
    dydxExact.append( 2*a*xPlot[n] + b )
plt.scatter(xPlot[0:N-1],dydxFD,label='dydx FD')
plt.plot(xPlot[0:N-1],dydxExact,'r',label='dydx exact')
plt.xlabel('x')
plt.ylabel('dydx')
plt.grid()
plt.legend()
plt.show()
```



In conclusion, the FD approximation appears to be a relatively accurate approximation, but

Q: What is the source of the small, constant difference between the curves?

1.6.4 Newton's method

Iterative processes are common to many useful scientific computing procedures - consider using **Newton's method** to find the roots of our quadratic equation $y = ax^2 + bx + c = f(x)$. If we take an initial guess the one root is found near $x_0 = -0.5$ where the subscript 0 refers to the iteration number, a more accurate prediction of a root location is

$$x_1 = x_0 - \frac{f(x_0)}{df/dx|_{x_0}} \qquad \text{with } df/dx|_{x_0} = 2ax_0 + b$$

Implementing this as a Python script:

[69]:
```
a,b,c = 1,3,2
x = -0.5
minError = 1e-6

for i in range(10):
```

```
    f = a*x**2 + b*x + c
    df = 2*a*x + b
    if df != 0:
        update = -f/df
        print('x0 = {:.4f}, update = {:.4e}'.format(x,update))
    else:
        print('infinite update for x = ',x)
        break # the for loop
    x += update
    if abs(update) < minError:
        print('\nconverged at x = {:.8f}'.format(x))
        break
```

```
x0 = -0.5000, update = -3.7500e-01
x0 = -0.8750, update = -1.1250e-01
x0 = -0.9875, update = -1.2348e-02
x0 = -0.9998, update = -1.5242e-04
x0 = -1.0000, update = -2.3231e-08

converged at x = -1.00000000
```

Q: Can the finite difference approximation to the quadratic equation be used in the Newton method instead of the exact derivative?

1.6.5 Case study: Nonlinear differential equation solution using Euler's method

Another common use for iterative algorithms is in the numerical integration of ordinary differential equation models. Consider, for example, the time rate of change of the concentration c_A, c_B in mol m^{-3} of liquid species A and B in a continuous stirred tank reactor (CSTR) with constant volume V in m^2, feed and product volumetric flowrates q in mol s^{-1} with reaction

$$2A \quad \rightarrow \quad B$$

at a rate $r = kc_A^2$ with rate constant k having units m^3 mol−1 s^{-1}. We assume the reaction has no effect on the mixture density. A material balance on species A gives

$$V\frac{dc_A}{dt} = q(c_{A,feed} - c_A) - Vkc_A^2$$

Dividing through by q, defining dimensionless time as $\tau = t/(V/q)$ and dimensionless concentration as $x = c_A/c_{A,feed}$ gives

$$\frac{dx}{d\tau} = 1 - x - \kappa x^2 \quad \text{subject to initial condition } x(\tau = 0) = 1$$

Q: Is κ dimensionless?

For this example, we take the dimensionless rate constant as $\kappa = 1$. Starting at our initial condition $x_0 = 1$ (subscript 0 denotes the initial time, subscript 1 the next point in time, etc.) and approximating our derivative with the *finite-difference* approximation, we derive our **forward Euler**

scheme with a **fixed** $\Delta\tau = \tau_1 - \tau_0$ as

$$\frac{dx}{d\tau} \approx \frac{x_1 - x_0}{\Delta\tau} = 1 - x_0 - \kappa x_0^2 \tag{1}$$

$$\text{so } x_1 = x_0 + \Delta\tau\left(1 - x_0 - \kappa x_0^2\right) \tag{2}$$

$$\text{so } x_2 = x_1 + \Delta\tau\left(1 - x_1 - \kappa x_1^2\right) \tag{3}$$

$$\vdots \tag{4}$$

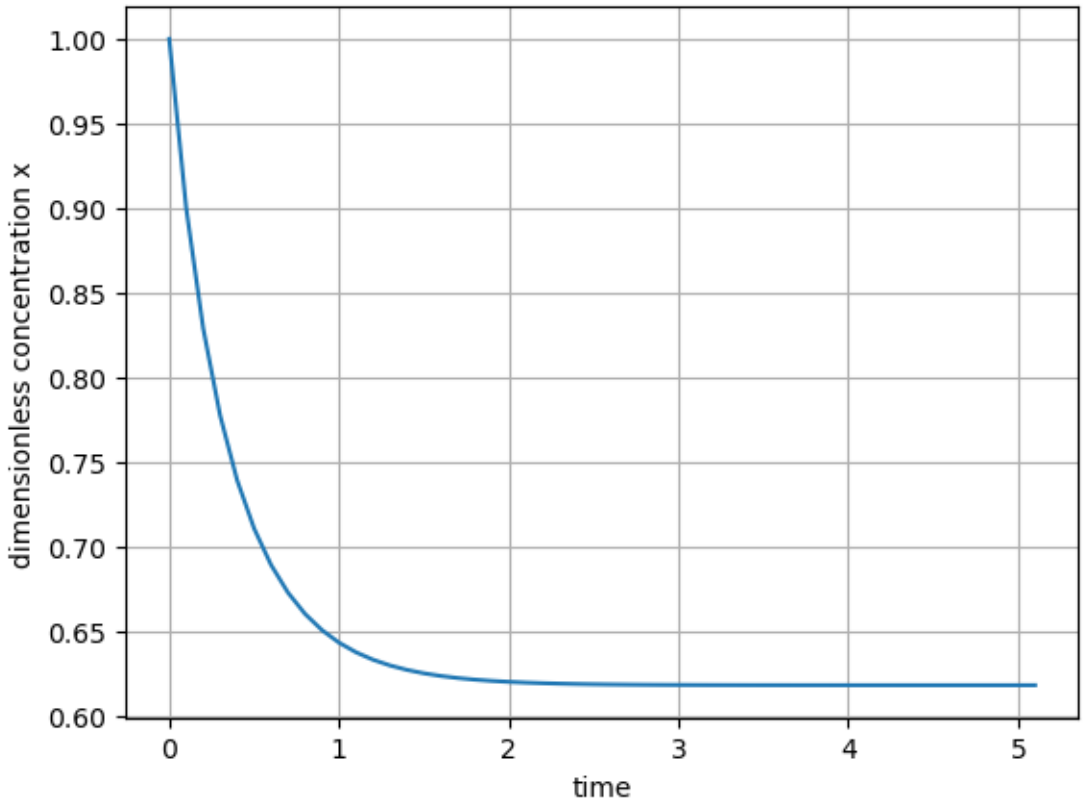
Note how by *bootstrapping*, we can step forward in time τ_k, $k = 0, 1, 2, 3, ...$

Implementing this as a Python script

[70]:
```
import math
Delt = 0.1 # small values of Delta t improve the stability
tStop = 5 # dimensionless time when the integration will stop
t,x = 0,1 # the initial condition
xPlot,tPlot = [x],[t]
kappa = 1

while t < tStop:
    update = Delt*( 1 - x - kappa*x**2 )
    t += Delt; x += update
    tPlot.append(t); xPlot.append(x)

print('at t = {:.2f}, x = {:.2f}'.format(t,x))
```

```
at t = 5.10, x = 0.62
```

[71]:
```
import matplotlib.pyplot as plt
plt.plot(tPlot,xPlot)
plt.xlabel('time')
plt.ylabel('dimensionless concentration x')
plt.grid()
plt.show()
```



Note how the dimensionless concentration x approaches a steady state value of

$$x_{stst} = \frac{1 - \sqrt{1 + 4\kappa}}{-2\kappa}$$

[72]:
```
print( 'x_stst = ',round( (1-math.sqrt(1+4*kappa))/(-2*kappa),3 ) )
```

```
x_stst =  0.618
```

Q: Why?

1.7 Functions

Functions are key to translating algorithms to resuable program segments in a computationally "safe" manner by limiting the **scope** of variables defined within the function. To illustrate this and the basic format of a function, consider the following simple exponentiation function:

[73]:
```
def expo(x):
    cube = x**3
    return cube
```

In this function, “expo” is the function name, “x” is the input parameter, “cube” is a variable local to the function that is returned from the function. Like “if” statements and loop structures, a 4 white-space indentation defined the function contents - note that the function will generate an error if “:” does not end the first line (of course comments can follow). To demonstrate functions in action, consider the results of the following script:

[74]:
```
p = 3
q = expo(p)
print(p,q)

print( 'p' in globals() ) # check if 'p' is a global variable
print( 'q' in globals() )
print( 'cube' in globals() )
```

```
3 27
True
True
False
```

We observe two important function properties: - That the variable names used to call a function or return a value need not be the same as in the function definition - Variables defined within a function, unless declared global, are local to the **scope** or **namespace** of the function - The return statement is not a function, but a keyword that allows the return of scalars (as in this case), lists, dictionaries, and any other object

This distiction between local and global variables can result in unexpected results – consider:

[75]:
```
p = 3
n = 4

def expo(x):
    cube = x**n
    return cube

n = 2
print(expo(p))
```

```
9
```

Before we leave this example, we examine how **default function input parameters** are defined – consider:

[76]:
```
def expo(x=1):
    return x**3

print( expo() ) # expo() calls the function using its default input parameter
print( expo(5) ) # overwrites the default parameter value
```

```
1
125
```

For functions with more than one input parameter, but only some have default values, it is generally better to list thouse that do not have default values first, followed by those with defaults. This means that if we have

def newFunc(a,b,c,d=1,e=2,f=3):

we can call the function in an assortment of valid ways:

x = newFunc(10,20,30) # defines a,b,c by user input, defaults to d,e,f x = newFunc(10,20,30,40) # replaces the default of input d x = newFunc(a=10,b=20,c=30,d=40) # same as above x = newFunc(10,20,30,f=0) # replaces the default of input f

1.7.1 lambda functions

For generally scalar, simple functions, a compact function notation is provided by the **lambda function** which has the general form

fn = lambda arguments : expression # note that imports are global

Consider computing the length of the 2D vector:

[77]:
```
h = lambda x,y : math.sqrt( x**2 + y**2 )

print('hypotenuse =',h(3,4))
```

```
hypotenuse = 5.0
```

Note the lack of a 4-whitespace indetation in the definition of the function.

Case study: interpolation of a physical property From the NIST Chemistry Webbook, we find that the heat capacity (C_p in J mol^{-1} K^{-1}) of methane gas at the two temperatures T (in K) is

[78]:
```
T = [100, 500] # K
Cp = [33.28, 46.63] # J mol^-1 K^-1
```

What is its value $C_{p,int}$ at $T_{int} = 300$ K? To determine the value, we will interpolate between the two data points:

$$C_{p,int} = \frac{C_p(500) - C_p(100)}{T(500) - T(100)}[T_{int} - T(100)] + C_p(100)$$

[79]:
```
CpInterp = lambda T,Cp,Tint : (Cp[1]-Cp[0])/(T[1]-T[0])*(Tint-T[0]) + Cp[0]
print( 'Cp =',round(CpInterp(T,Cp,300),2),'J mol^-1 K^-1' )
```

```
Cp = 39.95 J mol^-1 K^-1
```

compared to the true (NIST) value of $C_p = 35.76$ J mol^{-1} K^{-1}. So our interpolated estimate is in the correct range, but the accuacy is questionable.

Q: Why?

1.7.2 Modules

A module is a plain text file containing one or more Python functions with filename "moduleName.py" that are imported via "import moduleName" provided the correct **path** has been included (more on this in the next section). Consider the ideal gas module

[80]:
```
"""
Ideal gas properties
"""
kB = 1.3806503e-23 # Boltzmann constant, J/K
Avo = 6.0221413e23 # Avogadro's number mol^-1
R = kB*Avo         # gas constant, J/mol/K
Po = 101325        # Pa
To = 273.15        # K

def molarDens(T=To,P=Po,verbose=0):
    # ideal gas molar density, mol m^-3
    # input T is in K, P in Pa
    m = P/(R*T)
    if verbose:
        print('Molar density: {:.3e} mol m^-3'.format(m))
    else:
        return m

def meanFreePath(hsDiam,T=To,P=Po,verbose=0):
    # ideal gas mean free path, m
    # input hsDiam in m, T is in K, P in Pa
    m = R*T/( math.sqrt(2)*math.pi*hsDiam**2*Avo*P)
    if verbose:
        print('Mean free path: {:.3e} m'.format(m))
    else:
        return m

def wallCollisionRate(atM,T=To,P=Po,verbose=0):
    # ideal gas wall collision rate, m^-2 s^-1
    # input atM in kg, T is in K, P in Pa
    w = P/math.sqrt( 2.0*math.pi*atM*kB*T )
    if verbose:
        print('Wall collision rate: {:.3e} m^-2 s^-1'.format(w))
    else:
        return w
```

noting that the text between the " " " are interpreted as comments. We now demonstate its use

[81]:
```
T = 300 # K
P = 101325 # 1 atm in Pa
molarDens(T,P,True)
```

```
Molar density: 4.062e+01 mol m^-3
```

[82]:
```
# For argon:
hsDiam = 340e-12 # m
atM = 39.948/1000/Avo # kg
meanFreePath(hsDiam,T,P,True)
wallCollisionRate(atM,T,P,True)
```

```
Mean free path: 7.959e-08 m
Wall collision rate: 2.439e+27 m^-2 s^-1
```

1.7.3 Paths

Python knows where to find built-in functions and the modules that came with the distribution or added through the **pip** module manager. However, consider the situation where you are working on a Python project in the form of a large script found on your desktop. You would like to import the solar energy module "pysolray.py" from the directory "solarToolbox," a folder on your desktop.

Without any additional information, your Python script will not be able to locate the module - to tell Python where to look, we use the "addpath" function and then import the module:

[83]:
```
import sys
sys.path.append("solarToolbox")
import pysolray
```

Note the lack of output - this indicates that the new path to "pysolray.py" was successfully added and that the module was imported.

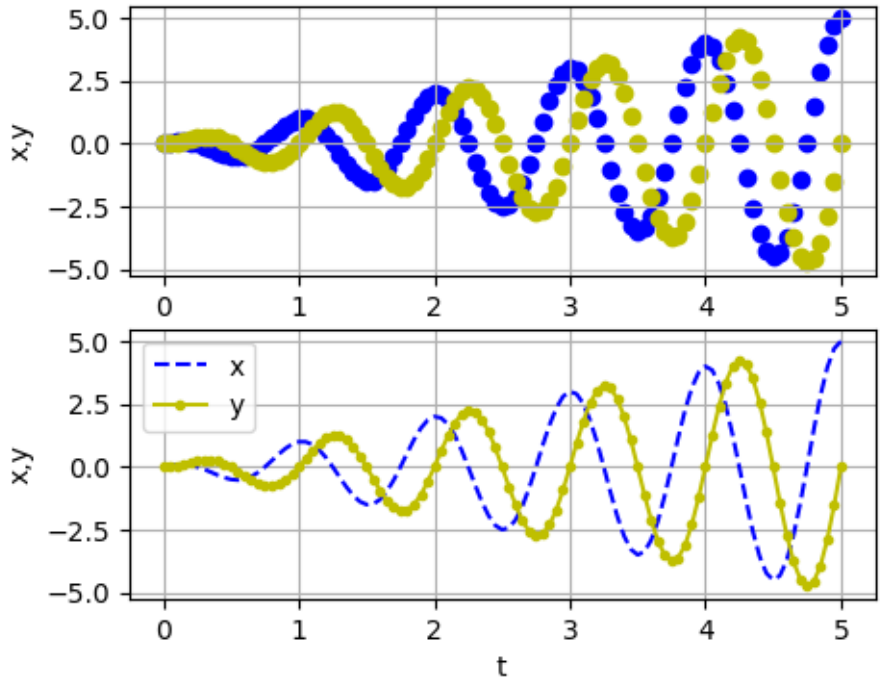
1.8 Plotting and visualization

Previously, we demonstated the use of "import matplotlib.pyplot as plt" to generate simple plots; we now turn to more advanced plotting options.

Subplots In our previous example, after defining lists x and y, we plotted the data with the equivalent of "plt.plot(x,y); plt.show()" We can gain more control over our plots by first defining a *figure*

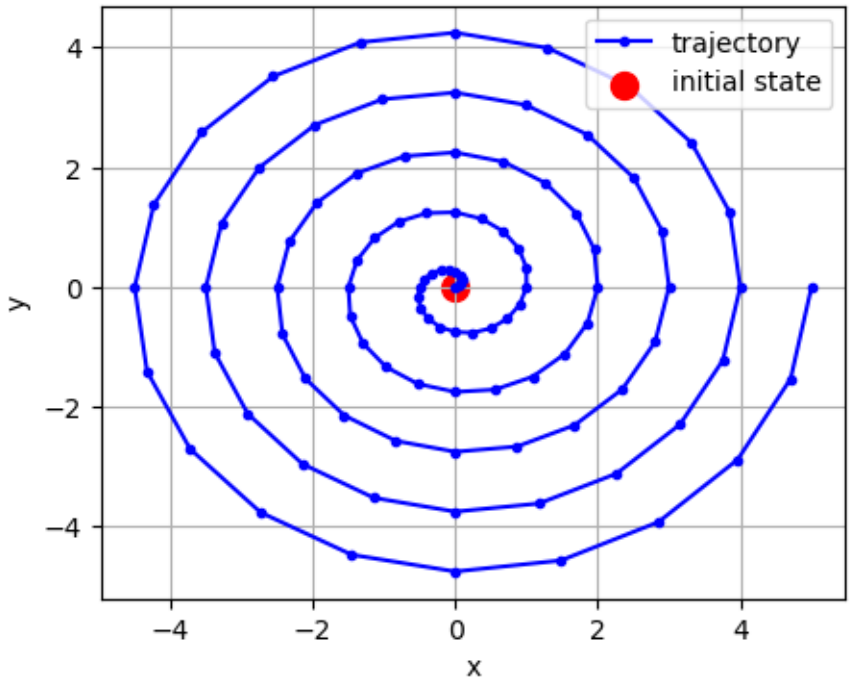
[84]:
```
t = numpy.linspace(0,5,101) # 101 points in [0,5]
x = t*numpy.cos(2*math.pi*t) # note element-by-element product *
y = t*numpy.sin(2*math.pi*t)

plt.figure(figsize=(5,4)) # create a 5in by 4in plot
plt.subplot(2,1,1) # subplot 1 of a 2x1 set of subplots
plt.scatter(t,x,color='b')
plt.scatter(t,y,color='y')
plt.ylabel('x,y')
plt.grid()
```

```
plt.subplot(212) # note lack of commas
plt.plot(t,x,'b--',label='x')
plt.plot(t,y,'y.-',label='y')
plt.xlabel('t')
plt.ylabel('x,y')
plt.grid()
plt.legend() # note that legend position can be specified
plt.show()
```



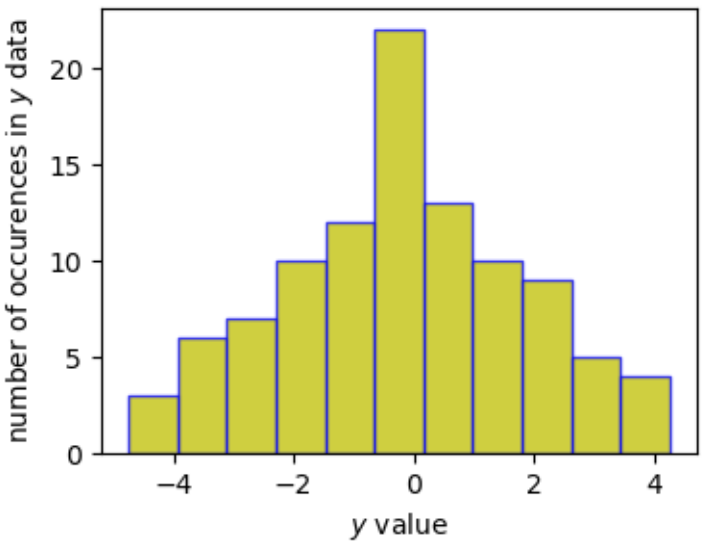
Phase-plane plots We also will find it valuable to plot the dynamics of low-dimensional systems in the **phase plane** where we can observe

[85]:
```
plt.figure(figsize=(5,4))
plt.plot(x,y,'b.-',label='trajectory')
plt.scatter(x[0],y[0],color='r',s=100,label='initial state') # s = size
plt.xlabel('x')
plt.ylabel('y')
plt.grid()
plt.legend()
plt.show()
```



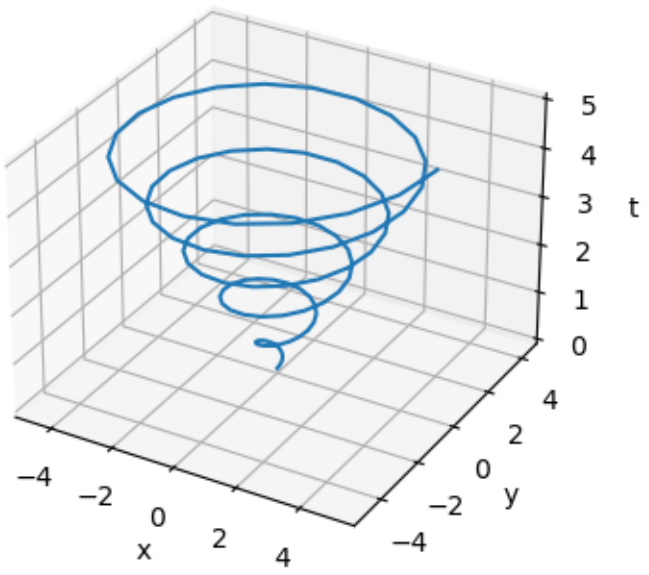
Creating a histogram For the discretely spaced trajectories we've been plotting, can we place the Y data into 11 evenly spaced bins and, using a bar chart, display the number of points in each bin?

[86]:
```
# a histogram of the data
plt.figure(figsize=(4,3))
n,bins,patches = plt.hist(y,11,facecolor='y',alpha=0.75, \
                          edgecolor='b')
plt.xlabel('$y$ value') # note math notation
plt.ylabel('number of occurences in $y$ data')
plt.show() # not needed in jupyter nb, otherwise required
```



3D plots For plotting in 3D, things are a bit more complicated as seen in plotting this 3D curve

[87]:
```
ax = plt.figure().add_subplot(projection='3d')
ax.plot(x,y,t)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('t')
ax.set_box_aspect(None,zoom=0.85) # prevents cutoff of z-axis label
plt.savefig('3Dplot.pdf') # save the figure to file 3Dplot.pdf
plt.show()
```



We note the mixed use of "ax." and "plt." functions where the former apply to a subset of the entire figure (in this case a (111) subplot - see the ".add_subplot(projection='3d')") while the "plt." operations apply to the entire figure.

Case study: NH3(T,P) equilibrium pcolor() plot We now consider the (exothermic) ammonia synthesis reaction that takes place in the Haber-Bosch process

$$\frac{3}{2}\mathrm{H}_2 + \frac{1}{2}\mathrm{N}_2 \quad \rightleftarrows \quad \mathrm{NH}_3 - 46 \text{ kJ mol}^{-1}$$

with process conditions $P \approx 200$ atm, $T \approx 450^o$ C. Under the assumption of ideal gas conditions, we can write the *equilibrium* relationship in terms of each component mole fraction y as

$$\frac{y_{NH_3}}{y_{H_2}^{3/2} y_{N_2}^{1/2}} = \left(\frac{P}{P^o}\right) K_{eq}$$

Shacham and Brauner provide the original (and corrected) equilibrium constant developed by Haber:

$$\log_{10} K_{eq} = 2.10 + \frac{9591/T - 4.6 \times 10^{-4} T + 8.5 \times 10^{-7} T^2}{4.571} - \frac{4.98}{1.985} \log_{10} T$$

where T is in K.

Let's translate this into a Python function of temperature T in K and P in atm:

[88]:
```
def NH3eqK(T,P):
    """
    Ammonia synthesis reaction equibrium constant calculations where
    the returned value K includes the pressure effect (P/Po)^nu,
    with temperature T in K, pressure P in atm. Assumes Po is 1 atm.
    """
    term0 = 2.10
    term1 = ( 9591/T - 4.6e-4*T + 8.5e-7*T**2 )/4.571
    term2 = 4.98*math.log10(T)/1.985
    Keq = 10**(term0+term1-term2)
    return P*Keq # based on Po = 1 atm
```

[89]:
```
# note how we can extract information about the function
help(NH3eqK)
```

```
Help on function NH3eqK in module __main__:

NH3eqK(T, P)
    Ammonia synthesis reaction equibrium constant calculations where
    the returned value K includes the pressure effect (P/Po)^nu,
    with temperature T in K, pressure P in atm. Assumes Po is 1 atm.
```

Let's try the function at room temperature (298 K) and pressure (1 atm):

[90]:
```
T = 298 # K
P = 1 # atm
print( 'K = {:.3e}'.format(NH3eqK(T,P)) )
```

```
K = 8.322e+02
```

which highly favors product (NH_3) formation; however, the reaction rate is known to be low at low temperatures, so let's raise T to an industrially relevant value T_{ind}:

[91]:
```
Tind = 298+450 # K
P = 1 # atm
print( 'K = {:.3e}'.format(NH3eqK(Tind,P)) )
```

```
K = 5.294e-03
```

and so we see that this greatly reduces the conversion to NH_3; therefore, let's raise the system to industrial operating pressure P_{ind}:

[92]:
```
Tind = 298+450 # K
Pind = 200 # atm
print( 'K = {:.3e}'.format(NH3eqK(Tind,Pind)) )
```

```
K = 1.059e+00
```

which appears to be substantially better - more interpretation will take place later in this course. For now:

Q: If we start with a stoichiometric mixture of H_2 and N_2 (e.g., 1 mole of N_2 and 3 moles of H_2), what will be the resulting mole fractions at equilibrium under the industrial operating conditions?

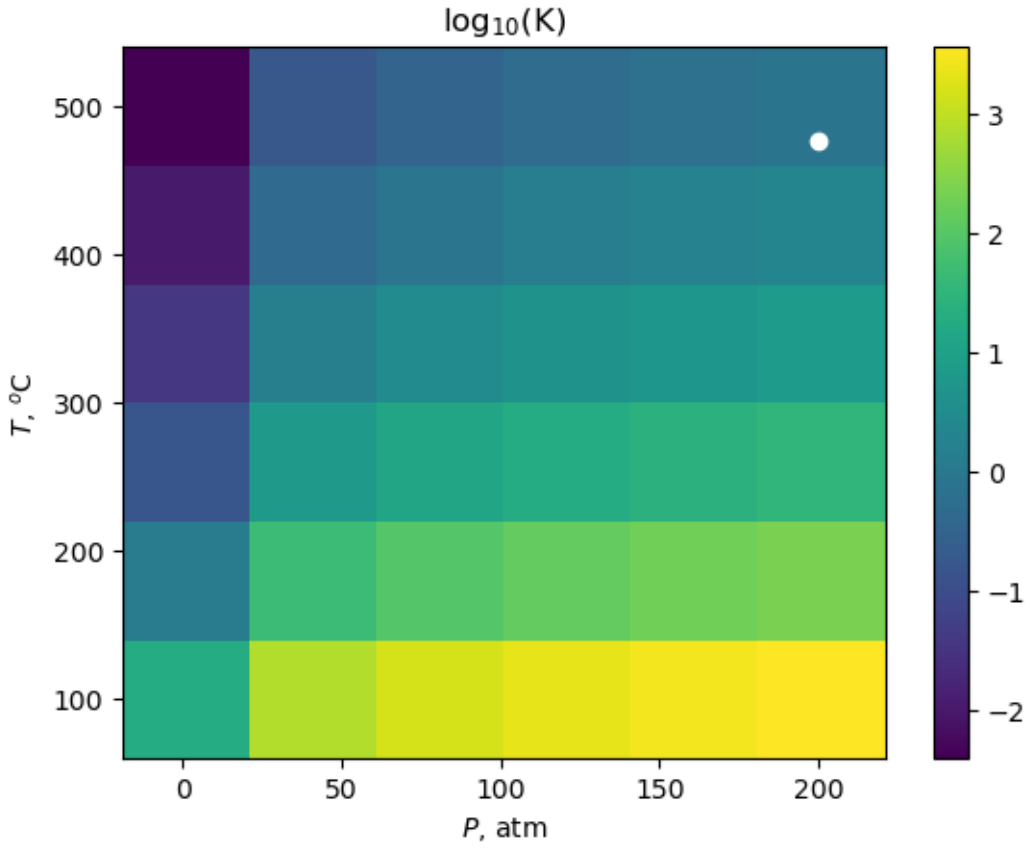
Let us wrap up with a plot of $K(T, P)$ over a range of input parameters

[93]:
```
T = numpy.linspace(100,500,6) # T values, deg C
P = numpy.linspace(1,201,6) # P values, atm
# set up grid arrays
Tplot = numpy.zeros((len(T),len(P)))
Pplot = numpy.zeros((len(T),len(P)))
Kplot = numpy.zeros((len(T),len(P)))
# fill the arrays
for i in range(len(T)):
    for j in range(len(P)):
        Tplot[i][j] = T[i]
        Pplot[i][j] = P[j]
        Kplot[i][j] = math.log10( NH3eqK(T[i]+273,P[j]) ) # log10 to reduce␣
 ↪range
print('Tplot:\n',Tplot)
print('Pplot:\n',Pplot)
print('Kplot (low T):\n',Kplot[0]) # print 1st row
print('Kplot high T):\n',Kplot[-1]) # print last row
```

```
Tplot:
 [[100. 100. 100. 100. 100. 100.]
 [180. 180. 180. 180. 180. 180.]
 [260. 260. 260. 260. 260. 260.]
 [340. 340. 340. 340. 340. 340.]
 [420. 420. 420. 420. 420. 420.]
 [500. 500. 500. 500. 500. 500.]]
Pplot:
 [[  1.  41.  81. 121. 161. 201.]
 [  1.  41.  81. 121. 161. 201.]
 [  1.  41.  81. 121. 161. 201.]
 [  1.  41.  81. 121. 161. 201.]
 [  1.  41.  81. 121. 161. 201.]
 [  1.  41.  81. 121. 161. 201.]]
Kplot (low T):
 [1.26166653 2.87445039 3.17015155 3.3444519  3.46849241 3.56486259]
Kplot high T):
 [-2.39819242 -0.78540856 -0.4897074  -0.31540705 -0.19136654 -0.09499636]
```

[94]:
```
fig, ax = plt.subplots()
c = ax.pcolor(Pplot,Tplot,Kplot) # note ordering of Pplot, Tplot
fig.colorbar(c, ax=ax) # add a colorbar
```

```
plt.scatter(Pind,Tind-272,color='w') # mark the industrial operating point
ax.set_xlabel('$P$, atm')
ax.set_ylabel('$T$, $^o$C')
ax.set_title('log$_{10}$(K)')
plt.show()
```



We can also generate the $P-T$ grid using the numpy.mgrid() function

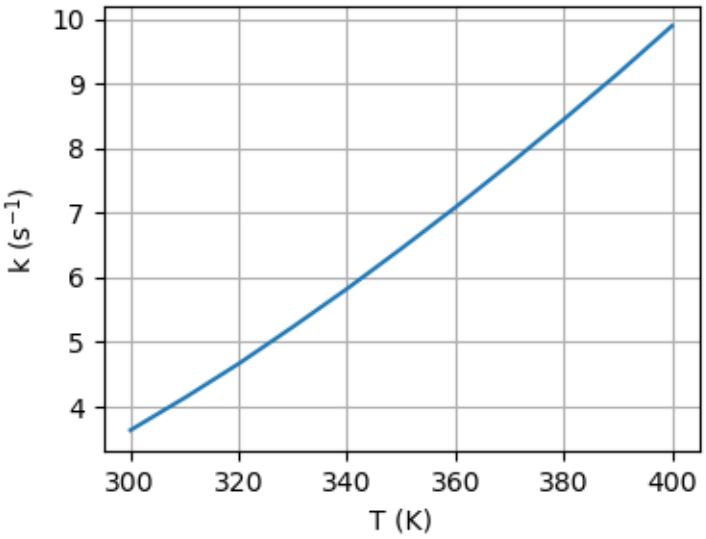
Reading data from external CSV files Frequently, data that we wish to analyze and plot are contained in a spreadsheet generated by another application, e.g., MS Excel. Let us consider a sampl .csv file containing reaction rate data for a 1st order reaction A → B:

[95]:
```
import csv
with open('rxnData.csv', 'r', newline='') as file:
    # create a csv.reader object
    csv_reader = csv.reader(file)
    # print each row in the CSV file
    T,k = [],[]
    for row in csv_reader:
```

```
        print(row)
        if row[0][0] != 'T': # removes title row
            T.append(float(row[0]))
            k.append(float(row[1]))
```

```
['T (K)', 'Rate k (s^-1)']
['300', '3.63']
['310', '4.13']
['320', '4.66']
['330', '5.23']
['340', '5.82']
['350', '6.44']
['360', '7.08']
['370', '7.75']
['380', '8.44']
['390', '9.15']
['400', '9.89']
```

[96]:
```
plt.figure(figsize=(4,3))
plt.plot(T,k)
plt.xlabel('T (K)')
plt.ylabel('k (s$^{-1}$)')
plt.grid()
plt.show()
```



Q: If $k = k_0 \exp(-E_a/RT)$, how do we determine k_0 and E_a?

[97]:
```
Tplot,Pplot = numpy.mgrid[100:500:80,1:201:40]
print('Tplot:\n',Tplot)
print('Pplot:\n',Pplot)
```

```
Tplot:
 [[100 100 100 100 100]
 [180 180 180 180 180]
 [260 260 260 260 260]
 [340 340 340 340 340]
 [420 420 420 420 420]]
Pplot:
 [[  1  41  81 121 161]
 [  1  41  81 121 161]
 [  1  41  81 121 161]
 [  1  41  81 121 161]
 [  1  41  81 121 161]]
```

1.9 Object-oriented Python programming

Up to now, all of the programming was *procedural* – programs consisting of step-by-step instructions to process data. **Object-oriented programming** puts the emphasis on the data by making possible definition of new *classes* with which *objects* are *instantiated* and maniputed using the *methods* of that class along with any methods that have been *inherited*.

1.9.1 Classes and objects

Consider some of the attributes that make up a person, such as name and age. We would also like to display these attributes corresonding to a individual object of class person, such as object "A" (think of it as variable A) as follows

A.name (attibute "name") A.age (attibute "age") A.display() (function or method "display")

[98]:
```
class person:
    def __init__(self):
        self.name = 'None'
        self.age = 0
    def display(self):
        print('Name:',self.name)
        print('Age:',self.age)
    def isTeen(self):
        print('Are they a teen?',self.age < 20 and self.age > 12)
```

[99]:
```
A = person() # create an object of person class
A.name = 'Ray' # explicitly set its name
A.age = 32 # explicitly set its age
print(A) # will not give as much information as expected
```

```
<__main__.person object at 0x7fdcf96f29d0>
```

There are a number of important points made in this simple example:

- A new class is created with the Python keyword "class" and the built-in function "def" (as with a function)
- Each class must have a *constructor method* (function) defined as **init**()
- The double underscore indicates "init" is a special method that is automatically called when a new instance (object) of that class is created; the init method is *not* called directly
- Each *method* of the class has as its first input parameter *self* which refers to the class
- For this example, the **default** name and age are "Bob" and 21, respectively
- Methods of the class are called using the objectName.method() format and attributes can be accessed as objectName.attributeName (note that objectName $\neq$ class name, but is the name of the instance (object)
- When a new instance is created, it is **crucial** that the class is called with (), e.g., A = person(), even when no parameters are passed to the constructor method.

Note that default attribute values can be assigned in the **init** input parameters:

[100]:
```
class person:
    def __init__(self,name="Jane",age=21):
        self.name = name
        self.age = age
    def display(self):
        print('Name:',self.name)
        print('Age:',self.age)
    def isTeen(self):
        print('Are they a teen?',self.age < 20 and self.age > 12)

B = person(age=13)
B.display()
B.isTeen()
```

```
Name: Jane
Age: 13
Are they a teen? True
```

1.9.2 Inheritance and polymorphism

Consider *deriving* a new, more specialized, (**child**) class from the (now **parent**) class we just defined to create a more specialized class of people: students

[101]:
```
class student(person):
    def __init__(self,name=' ',age=0,school=' '):
        self.name = name
        self.age = age
        self.school = school
    def display(self):
        print('Name:',self.name)
```

```
        print('Age:',self.age)
        print('School:',self.school)

C = student('Betty',19,'University of Maryland')
C.display()
C.isTeen()
```

```
Name: Betty
Age: 19
School: University of Maryland
Are they a teen? True
```

We observe a number of important Python object-oriented features at work in this simple example:

- The definition of this child class includes as an input parameter the name of the parent class from which it is derived in the class definition "class student(person):"
- The **methods** of the "person" class are **inherited** by the child class definition - that is why the "isTeam" method does not have to be redefined
- However, to add the "school" attribute, we **overload** the "**init**" and "display" methods - this does not affect the behavior of those method in the parent class. This is **polymorphism** and is something we've seen before in the Python language, such as with the addition/concatination operature "+"

[102]:
```
print(2+2) # addition
print('a'+'b') # concatination
```

```
4
ab
```

Case study: The prUnit and prStream classes Let us consider creating a simple set of chemical process flowsheet process units and the streams that connect them. We will define classes for the process streams and a generic process unit. Three more specific processes in the form of a mixer, reactor, and separator will be derived from the "prUnit" class.

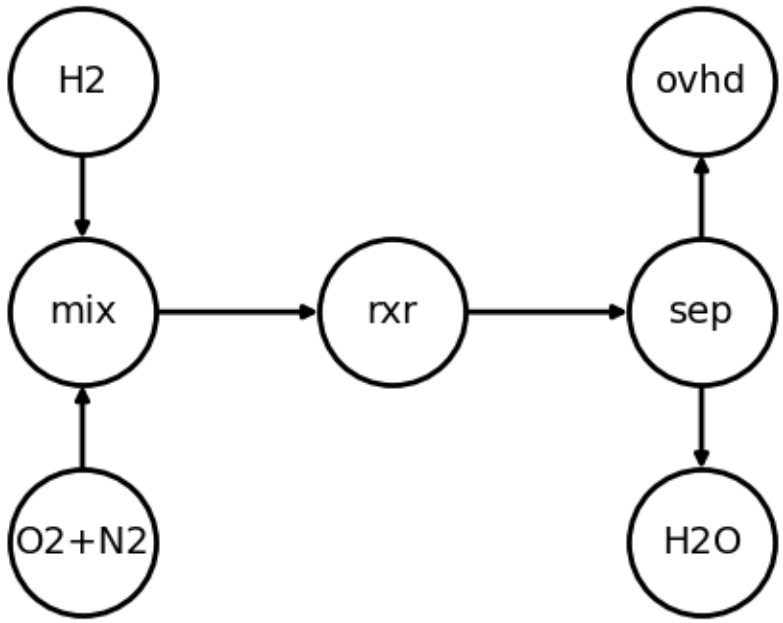
Our objective in this case study is to study the reaction

$$2H_2(g) + O_2(g) \quad \rightarrow \quad 2H_2O(l)$$

with the following mixing, reaction, and separation process

[103]:
```
import networkx as nx

nodes = ['H2','O2+N2','mix','rxr','sep','ovhd','H2O']
G = nx.DiGraph()
G.add_edge('H2', 'mix')
G.add_edge('O2+N2','mix')
G.add_edge('mix', 'rxr')
G.add_edge('rxr', 'sep')
G.add_edge('sep', 'ovhd')
G.add_edge('sep', 'H2O')
```

```
# explicitly set positions
pos = {'H2': (-1, 1), 'O2+N2': (-1, -1), 'mix': (-1, 0), 'rxr': (0, 0), 'sep':␣
 ↪(1, 0), 'ovhd': (1, 1), 'H2O': (1,-1)}

options = {
    "font_size": 14,
    "node_size": 3000,
    "node_color": "white",
    "edgecolors": "black",
    "linewidths": 2,
    "width": 2,
}
nx.draw_networkx(G, pos, **options)
# nx.draw_networkx_nodes(G, pos, nodelist=[1,2,6,7], node_shape='o',␣
 ↪node_color='blue')
# nx.draw_networkx_nodes(G, pos, nodelist=[3,4,5], node_shape='s',␣
 ↪node_color='red')

# Set margins for the axes so that nodes aren't clipped
ax = plt.gca()
ax.margins(0.20)
plt.axis("off")
plt.show()
```



[104]:
```
"""
Module flowSheet.py
"""
class prStream:
    """
    Chemical process stream class; defines stream composition and molar
    flow rates
    """
    def __init__(self,species=[],molFlow=[],name="strm"):
        self.species = species # chemical species name list for the process
        self.molFlow = molFlow # molar flow list (mol s^-1); same order as␣
 ↪species
        self.name = name # stream name
    def display(self):
        print('Stream name:',self.name)
        for i in range(len(self.species)):
            print('  ',self.species[i],':',self.molFlow[i],'mol s^-1')

class prUnit:
    """
    Generic process unit with multiple feed streams (in the form of a
    prStream list) and multiple product streams
    """
    def __init__(self,streamIn,name="proc",tranStoich=[]):
        self.name = name # process unit name
        if isinstance(streamIn,list):
            self.streamIn = streamIn # list of prStream objects
        else:
            self.streamIn = [streamIn] # list of prStream objects
        self.tranStoich = tranStoich # feed/product transition array
        self.streamOut = [] # list of product streams
    def display(self):
        print('Process unit name:',self.name)
        print('Process feed streams:')
        for i in range(len(self.streamIn)):
            self.streamIn[i].display()
        print('Transformation stoichiometry:',self.tranStoich)
        if len(self.streamOut) > 0:
            print('Product streams:')
            for i in range(len(self.streamOut)):
                self.streamOut[i].display()

class prMix(prUnit):
    """
    Multi-feed mixing unit producing one combined product prStream
    """
```

```
    def mix(self,streamOutName='mixProd'):
        species = self.streamIn[0].species
        outFlow = [0]*len(species)
        for i in range(len(self.streamIn)):
            streamMolFlow = self.streamIn[i].molFlow
            for j in range(len(species)):
                outFlow[j] += streamMolFlow[j]
        self.streamOut = [prStream(species,outFlow,streamOutName)]

class prRxr(prUnit):
    """
    Single feed/product reactor class - note that a single list of
    tranStoich must be specified when instantiating a prRxr object
    """
    def react(self,conversion=1,streamOutName='rxrProd'):
        species = self.streamIn[0].species
        molFlow = self.streamIn[0].molFlow
        extentLimit = 1e6
        for i in range(len(species)):
            if self.tranStoich[i] < 0: # a reactant
                extent = molFlow[i]/abs(self.tranStoich[i]) # note abs()
                if extent < extentLimit:
                    extentLimit = extent
        outFlow = []
        for i in range(len(species)):
            molChange = self.tranStoich[i]*extentLimit*conversion
            outFlow.append(molFlow[i]+molChange)
        self.streamOut = [prStream(species,outFlow,streamOutName)]

class prSep(prUnit):
    """
    Single feed, multi-product separator. The degree of separation
    is included in the instantiation of a prSep object in the
    tranStoich list:
    tranStoich = [ [ coefficients for overhead stream ],
                   [ mid-level stream(s) ],
                   [ bottoms product ] ]
    """
    def sep(self,streamOutName='sepProd'):
        species = self.streamIn[0].species
        molFlow = self.streamIn[0].molFlow
        numbProdStrm = len(self.tranStoich)
        if not isinstance(self.tranStoich[0],list):
            numbProdStrm = 1
        self.streamOut = []
        for i in range(numbProdStrm):
            outFlow = []
```

```
            for j in range(len(species)):
                outFlow.append(molFlow[j]*self.tranStoich[i][j])
            self.streamOut.append(␣
 ↪prStream(species,outFlow,streamOutName+str(i)) )
```

[105]:
```
test = prStream(species=['H2', 'O2', 'N2', 'H2O'],molFlow=[5,2,8,4],)
S = prSep(test,tranStoich=[[1,1,1,0],[0,0,0,1]]) # each column sum = 1
S.sep()
S.display()
```

```
Process unit name: proc
Process feed streams:
Stream name: strm
   H2 : 5 mol s^-1
   O2 : 2 mol s^-1
   N2 : 8 mol s^-1
   H2O : 4 mol s^-1
Transformation stoichiometry: [[1, 1, 1, 0], [0, 0, 0, 1]]
Product streams:
Stream name: sepProd0
   H2 : 5 mol s^-1
   O2 : 2 mol s^-1
   N2 : 8 mol s^-1
   H2O : 0 mol s^-1
Stream name: sepProd1
   H2 : 0 mol s^-1
   O2 : 0 mol s^-1
   N2 : 0 mol s^-1
   H2O : 4 mol s^-1
```

[106]:
```
s = ['H2', 'O2', 'N2', 'H2O']
fA = [5, 0, 0, 0] # flow rates, mol s^-1
fB = [0, 2, 8, 0] # flow rates, mol s^-1
feed = [ prStream(s,fA,'FeedA'), prStream(s,fB,'FeedB')] # feed prStream objects
M = prMix(feed,'Mixer')
M.display()
```

```
Process unit name: Mixer
Process feed streams:
Stream name: FeedA
   H2 : 5 mol s^-1
   O2 : 0 mol s^-1
   N2 : 0 mol s^-1
   H2O : 0 mol s^-1
Stream name: FeedB
   H2 : 0 mol s^-1
   O2 : 2 mol s^-1
```

```
   N2 : 8 mol s^-1
   H2O : 0 mol s^-1
Transformation stoichiometry: []
```

[107]:
```
M.mix()
M.display()
```

```
Process unit name: Mixer
Process feed streams:
Stream name: FeedA
   H2 : 5 mol s^-1
   O2 : 0 mol s^-1
   N2 : 0 mol s^-1
   H2O : 0 mol s^-1
Stream name: FeedB
   H2 : 0 mol s^-1
   O2 : 2 mol s^-1
   N2 : 8 mol s^-1
   H2O : 0 mol s^-1
Transformation stoichiometry: []
Product streams:
Stream name: mixProd
   H2 : 5 mol s^-1
   O2 : 2 mol s^-1
   N2 : 8 mol s^-1
   H2O : 0 mol s^-1
```

[108]:
```
p = prRxr(streamIn=M.streamOut[0],tranStoich=[-2, -1, 0, 2],name="Rxr")
p.display()
```

```
Process unit name: Rxr
Process feed streams:
Stream name: mixProd
   H2 : 5 mol s^-1
   O2 : 2 mol s^-1
   N2 : 8 mol s^-1
   H2O : 0 mol s^-1
Transformation stoichiometry: [-2, -1, 0, 2]
```

[109]:
```
p.react(conversion=1)
p.display()
```

```
Process unit name: Rxr
Process feed streams:
Stream name: mixProd
   H2 : 5 mol s^-1
   O2 : 2 mol s^-1
   N2 : 8 mol s^-1
```

```
   H2O : 0 mol s^-1
Transformation stoichiometry: [-2, -1, 0, 2]
Product streams:
Stream name: rxrProd
   H2 : 1.0 mol s^-1
   O2 : 0.0 mol s^-1
   N2 : 8.0 mol s^-1
   H2O : 4.0 mol s^-1
```

1.9.3 UML diagrams

UML - the **Unified Modeling Language** - is a visual (not programming) language that is useful for understanding the relationships between elements of a software package. For example, object classes are represented by rectangles divided into segments listing object attributes and methods, as well as the inheritance relationships as arrows pointing **to** the parent (super) class **from** the child (sub/derived) classes. For example, we can view the relationship between the flowSheet classes, visualized using **pylint.pyreverse** (a static Python code analysis package):

[110]:
```
import flowSheet
import pylint.pyreverse as prev # are these imports necessary if the Unix␣
 ↪commands are used?
```

[111]:
```
! ls # the ! enables execution of Unix commands
! pyreverse -o html flowSheet.py
! open classes.html
```

```
__pycache__             flowSheet.py            rateData.py
3Dplot.pdf              linearSystems.ipynb     rxnData.csv
archive                 nonlinearSystems.ipynb  solarToolbox
classes.html            pythonIntro.ipynb       stateSpace.ipynb
parsing flowSheet.py…
```

This also can be run outside of Jupyter notebooks by typing at the shell prompt:

pyreverse -o html flowSheet.py

and then opening classes.html in a browser

Note it is important that graphviz for Python is installed - if not, at the Unix prompt: pip install graphviz

[ ]:
```

```