Python Introduction

January 15, 2024

1 Jupyter notebook

Welcome to Jupyter notebook! Most probably many of you have used this before. Jupyter notebook is an interface to iPython ("interactive Python") that is especially convenient for teaching and illustration. The utility of Jupyter notebooks lies in the ability to intermingle code, text (including LaTeX rendering for math), and graphics in a single coherent narrative. A Jupyter notebook file can be opened in a web browser and subsequently exported as a PDF file to give it the appearance of a written document. This might be useful for your homework assignments. Bear in mind that Jupyter is not always the best choice - when developing a large base of code rather than just snippets, it can be more trouble than it is worth. For those cases, an IDE (integrated development environment) is superior. One called "Spyder" is packaged with the Anaconda distribution.

To run a code block in Jupyter, click it and press the shift and enter keys at the same time.

This introduction is far from exhaustive - if you feel you need more help getting up to speed, Harvard SEAS offers Python bootcamp videos accessible at https://canvas.harvard.edu/courses/95581?ct=t(SEAS_Career_Grad_09_05_2022_01). Of course on the internet more generally there are more resources than we could even hope to begin to mention here.

2 Basic Python concepts

2.1 Variables

We can assign to a variable with an equals sign as follows.

[1]:
```
# This is a comment! Note that comments begin with the # sign.
# Please use many of these in your assignments to help us understand your␣
↪thinking.

x = 5 # assign 5 to x

x = x + 1 # add 1 to x
x += 1 # shorthand for the above, also applies to -, *, and /

y = x**2 # exponentiation in Python is denoted by **, not ^ as in other␣
↪languages
```

We can always use print to view the value of a variable or to print a string. In the following, we use the convenient feature of Python that strings can be concatenated by addition.

[2]:
```
print('Hello, AP 217! The value of x is ' + str(x) + '.')
print('And the value of y is ' + str(y) + '.')
```

```
Hello, AP 217! The value of x is 7.
And the value of y is 49.
```

2.2 Data types

The basic numeric types in Python are int, float, and complex.

Python takes care of the memory requirements for all of these, but when using numpy (see below) one can actually specify the memory footprint (bitdepth) of these (e.g., float32 for a 32-bit floating point number). Probably not necessary in this course, but an important consideration when optimizing code.

Below we assign each of these to variables.

[3]:
```
x = 5 # x is an integer
y = 5.33 # y is a floating point number
y = 5. # simply inserting the decimal with no more information is enough to␣
 ↪make y a float, not an int
type(y)
```

[3]:
```
float
```

Complex numbers are important to this course and optics in general since they denote phase. These can be assigned as follows:

[4]:
```
y = 3+4*1j # '1j' is the complex unit
```

[5]:
```
print(y)
print(abs(y))
```

```
(3+4j)
5.0
```

Booleans in Python are simply given by True and False. Based on these, we can construct conditional statements, often based on the comparators $>$, $>=$, $<$, $<=$, $==$, and !=.

[6]:
```
if 5<4:
    print("Math is broken!")
else:
    print("Math works!")

# compound logical statements can be formed with the 'and' and 'or' keywords
(4>3) and (3>15) or (3==4)
```

```
Math works!
```

[6]:
```
False
```

Lists are denoted by square brackets []. Bear in mind that in Python **list indexing begins at 0, not 1.**

[7]:
```
my_list = [5, 2, 3, 1, 4] # create a list of ints
first_element = my_list[0] # first element indexed with 0
last_element = my_list[-1] # we can also index from the end of the list
second_to_last_element = my_list[-2]

print('The list is ' + str(len(my_list)) + ' elements long.')
print('Its first element is: ' + str(first_element))
```

```
The list is 5 elements long.
Its first element is: 5
```

A data structure that is somewhat unique to Python is the *dictionary*. Dictionaries are encapsulated in curly braces {}. A dictionary contains 'keys' that point to 'values'. The keys must be of a consistent data type, but the values need not be.

Dictionaries in a sense are like lists, except that values can be accessed by the keys rather than a list index.

[8]:
```
# here is a sample dictionary for, e.g., tracking student grades
my_dict = {'Bob': 95, 'Susan': 98, 'Jerry': 72}

# easily recall Susan's grade
print(my_dict['Susan'])

# dictionaries can have mixed values, so long as the keys are of a consistent␣
↪data type (int in this case)
mixed_dict = {1: 'Hello', 2: 3, 67: 54.}
```

```
98
```

2.3 Iteration

Iteration in Python is provided by the for keyword as in the following.

[9]:
```
x = 1

# iterate for i = 0 through 9
for i in range(10):
    x += 1

print(x)
```

```
11
```

Note the range keyword to easily create a list of numbers between 0 and some value.

When filling the entries of a list using a for loop, Python provides a very nice simplification. For example, in the following we have a list of student names and we want to extract the first initial of

each.

[10]:
```
student_names = ['Bob', 'Susan', 'Jerry']

# extract first letter in each string using a nice Python list comprehension
first_initials = [name[0] for name in student_names]

print(first_initials)
```

```
['B', 'S', 'J']
```

2.4 Functions

When pieces of code are called over and over again, it is useful to wrap the repeated code as a *function*. Functions in Python are defined with the def keyword. Functions in Python can return a value, or not. The return keyword is used to define the function's output, if any.

Below is a function that squares any input and returns it. The name of the function is given, as well as the name of the input variables (just one in this case) which can be used throughout the function's body.

[11]:
```
def square(x):
    return x**2

q = square(2)
print(q)
```

```
4
```

Here is an example function that doesn't return anything.

[12]:
```
def sizer(y):
    if y > 5:
        print("That's a big variable!")
    elif y>0 and y<=5:
        print("That's a medium variable!")
    else:
        print("That's a small variable!")

x = 4
sizer(x)
```

```
That's a medium variable!
```

Functions in Python can have both *default* and *keyword* arguments. Default arguments are necessary inputs for using the function and must be provided when the function is called. Keyword arguments are necessary for the function to run, but the user does not necessarily have to provide their value when the function is called. Keyword arguments are denoted in the function header by a name and an equals sign. If the user provides their value, the provided value is used; if not, the value provided in the function definition is used.

```
[13]: # this function exponeniates x by y
      # x is a default argument, and y is an optional keyword argument
      def power_law(x, y=2):
          return x**y

      print(power_law(5))
      print(power_law(5, y=10))
```

```
25
9765625
```

It is "common courtesy" to others reading your code to provide nice documentation for your functions, including what they take in and what they output. You don't necessarily have to do it in one specific way for this class, but in software development there are standards for doing so. One example is shown for the function we just wrote:

```
[14]: def power_law(x, y=2):
          """
          A function to compute exponentiation

          Parameters
          ----------
          x : float or int
              the 1st param name `first`
          y :
              the power to which x is raised, defaults to 2

          Returns
          -------
          float or int
              x^n

          """
          # body of function comes after docstring above
          return x**y
```

For very simple functions, we can define an *anonymous* function which allows us to avoid a few lines of code.

2.5 Classes

A more advanced concept in Python, and programming in general, is that of classes. These form the basis of object-oriented programming, a key concept in software design. Object-oriented programming is probably not necessary for this course, where the emphasis is on code snippets and solving very specific problems. But it is included here for completeness, and if you spot an example in the course that might benefit from taking an object-oriented approach, don't hesitate to do so.

In the following we define a class that stores important information about a graduate student.

[15]:
```
# classes are defined by a name, by convention that is capitalized (while␣
↪functions are lower-case)
# classes are a convenient way to organize information and functions
# the self keyword below must be passed to a function for the function to have␣
↪access to the current instance of the class, the "self" to change things␣
↪about it

class GraduateStudent:

    # each class must have an __init__ method
    # this is the "constructor" method called by default when a new instance of␣
↪the class is created
    # in this case the constructor just takes in the student's name
    def __init__(self, name):
        self.name = name
        self.gpa = 4.0
        self.course_list = []

    # this is a method to add a course to the student's list
    def add_course(self, course_name):
        self.course_list.append(course_name)

    def modify_gpa(self, new_gpa):
        self.gpa = new_gpa

    def describe_student(self):
        print('Name: ' + self.name)
        print('Enrolled in: ' + str(self.course_list))
        print('GPA: ' + str(self.gpa))
```

Now we can interact with the class we just created:

[16]:
```
noah = GraduateStudent('Noah Rubin') # call the constructor method, instantiate␣
↪a new object of the class

noah.add_course("Optics for poets") # add a class
noah.modify_gpa(2.1)

noah.describe_student() # call the class method to learn about the student
```

```
Name: Noah Rubin
Enrolled in: ['Optics for poets']
GPA: 2.1
```

2.6 Package imports

One of the advantages of Python is the large volume of packages available, each easily imported with the import statement. Below we import one that will be of use in the next section.

[17]:
```
# import the package numpy, but allow us to refer to it in shorthand as 'np'
import numpy as np
```

3 Numpy

3.1 Creating numpy arrays

Python is a general-purpose programming language, used across all areas of software, including web development. However, what makes it useful for scientific computing specifically is the numpy package (and to a lesser extent, scipy). Numpy and its methods give Python enhanced ability to define, manipulate, and compute with arrays. Numpy and scipy also provide a number of optimized methods for numerical linear algebra, optimization, and other tasks.

We can define a numpy array structure from a simple list of numbers.

[18]:
```
sample_array = np.array([2,1,5,2]) # define numpy array from a list of numbers
sample_array.shape # query its shape, in this case just one number because the␣
↪array is 1D
```

[18]:
```
(4,)
```

Numpy arrays can have arbitrary dimensions (1, 2, 3, …, N). Here is a 3D array:

[19]:
```
three_d_array = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11,␣
↪12]]])
three_d_array.shape
```

[19]:
```
(3, 2, 2)
```

If we know the desired shape of a numpy array ahead-of-time, it is often easy to initialize it as either an array of zeros or an array of ones using built-in methods:

[20]:
```
sample_zeros = np.zeros([3,2,6]) # a 3x2x6 matrix full of zeros
sample_ones = np.ones([3,3]) # a 3x3 matrix full of ones
sample_ident = np.eye(10) # the 10x10 identity matrix with ones on diagonal;␣
↪this function only works for 2D arrays
```

Another important way of assembling arrays in numpy are the functions linspace and arange.

[21]:
```
# make an equally (linearly) spaced array with linspace
x = np.linspace(0, 10, 50) # 50 equally spaced values between 0 and 10 - use␣
↪linspace to control the length of the array

# values spaced by 0.1 between 0 and 10; use arange to control the element's␣
↪spacing, but not the total array length
y = np.arange(0, 10, 0.1)
```

3.2 Slicing

Once an array is created, one of the most important aspects of numpy arrays is *slicing*. Slicing refers to accessing parts of a given numpy array in a smart way. Slicing often uses the notation : and :: as a shorthand in ways that will be demonstrated below.

[22]:
```
# simple 1D slicing example
x = np.arange(0, 10, 1) # simple array of equally-spaced values between 0 and 9
x[2:-3] # a simple 1D example of slicing, take the 3rd element to the␣
 ↪3rd-to-last
x[:-2] # take the beginning of the array to the 2nd-to-last
x[::2] # take every 2nd element of the array
x[2::2] # take every 2nd element of the array, starting from the 3rd element
```

[22]:
```
array([2, 4, 6, 8])
```

[23]:
```
# multi-dimensional slicing examples
x = np.random.rand(5, 3, 2) # create a 5x3x2 array filled with random values

# all elements along first axis, only the elements in the 1st position along␣
 ↪the other axes
x[:, 0, 0]

# 2nd through 4th positions on 1st axis, beginning through 2nd-to-last element␣
 ↪of 2nd axis, every 2nd element along the 3rd axis
out = x[1:3, :-2, ::2]
out.shape
```

[23]:
```
(2, 1, 1)
```

3.3 Meshgrid

A convenient technique when working with coordinate systems is to use meshgrid. This method converts N 1D arrays that serve as coordinate axes into a list of N, N-dimensional arrays that store the coordinates of each point in the grid within. These can then be used to compute other quantities in a vectorized form.

[24]:
```
# define x, y, z positions along three axes
x, y, z = np.linspace(-5, 5, 5), np.linspace(-5, 5, 5), np.linspace(-5, 5, 5)

# apply the meshgrid command
X, Y, Z = np.meshgrid(x, y, z)
print(X.shape) # each output array is 3D

# now these can be used in vectorized computations
# for example, computing the distance of each point in the 3D space from the␣
 ↪origin
R = np.sqrt(X**2 + Y**2 + Z**2)
```

```
(5, 5, 5)
```

3.4 Matrix multiplication

Element-wise multiplication is denoted in Python by *. Matrix multiplication is denoted by @. **Don't mix them up!**

[25]:
```
# define two 2x2 matrices
A = np.array([[1, 1], [1, 1]])
B = np.array([[1, 2], [3, 4]])

# element-wise multiplication
print(A*B)

# actual matrix multiplication - the results are not the same!
print(A@B)
```

```
[[1 2]
 [3 4]]
[[4 6]
 [4 6]]
```

3.5 Sparse matrices

Sparse matrices refer to matrices for which the majority of the elements are 0. Sparse matrices appear throughout numerical methods in mathematics and the physical sciences. When an array is sparse, acknowledgment of its sparsity can, through a variety of numerical methods not discussed in this course (to learn more consider taking courses in applied mathematics), give significant computational advantage in solving linear algebra problems. Intuitively it is clear why - when the majority of the elements of an array are simply 0, only storing those which are not (rather than all $N \times N$ elements) is far more memory-efficient.

Scipy provides a library (scipy.sparse) providing special methods for constructing sparse numpy arrays and performing mathematics on them, such as solving linear systems of sparse arrays and finding their eigenvalues/vectors.

Two common methods for constructing sparse matrices are 1) Individually labeling the row, column, and entry of non-zero elements and, 2) Specifying individual diagonals of a matrix as 1D arrays and constructing the array from the sub-diagonals.

[26]:
```
import scipy.sparse as sparse

# create a sparse matrix in coo format (COOrdinate format)
row  = np.array([0, 3, 1, 0]) # rows of non-zero elements
col  = np.array([0, 2, 1, 3]) # columns of the non-zero elements
data = np.array([4, 5, 7, 9]) # the actual element values
mat = sparse.coo_matrix((data, (row, col)), shape=(4, 4))
mat.toarray() # display non-sparse version of array; never call on a very large␣
 ↪matrix or you will crash the Python console!
```

```
[26]: array([[4, 0, 0, 9],
             [0, 7, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 5, 0]])
```

```
[27]: N = 5 # side length of square matrix
diag_0 = np.ones(N)
diag_neg_2 = -2*np.ones(N-2)

# give the diagonals, their offset from the center diagonal, and the matrix␣
 ↪shape
mat = sparse.diags([diag_0, diag_neg_2], [0, -2], shape=[N, N])
mat.toarray()
```

```
[27]: array([[ 1.,  0.,  0.,  0.,  0.],
             [ 0.,  1.,  0.,  0.,  0.],
             [-2.,  0.,  1.,  0.,  0.],
             [ 0., -2.,  0.,  1.,  0.],
             [ 0.,  0., -2.,  0.,  1.]])
```

3.6 Optimization

Optimization is a complex topic. However, it is useful to understand the layout of an optimization problem and how one can be implemented in Python. There are entire Python packages dedicated to optimization routines. Scipy has some integrated into scipy.optimize.

Here, as a trivial example we will find the angle x that minimizes the function $\sin x$ such that x does not exceed 0.25. Of course, the building blocks here could be extended to analyze far more complicated, in general multi-variate objective functions in a real application.

The convenience function curve_fit from this same package is also of interest.

```
[28]: # minimize is a general purpose optimization function from scipy.
from scipy.optimize import minimize

# define the objective function to be minimized
def objective(x):
    return np.sin(x)

# initial guess for the optimum value
# because of the shape of the sine function, the initial guess dictates whether␣
 ↪the global minimum is actually found
x0 = np.pi

# inequality constraints in minimize are set up by providing a function that␣
 ↪needs to be >= 0
constraint = {'type': 'ineq', 'fun': lambda x: x-0.25}
```

```
# call the optimization
res = minimize(objective, x0, constraints=constraint)

print('Optimized x value: ' + str(res.x))
print('Minimized function value at x: ' + str(res.fun))
res.x # the optimized value of x
res.fun
```

```
Optimized x value: [4.71244431]
Minimized function value at x: [-1.]
```

[28]:
```
array([-1.])
```

4 Plotting with matplotlib

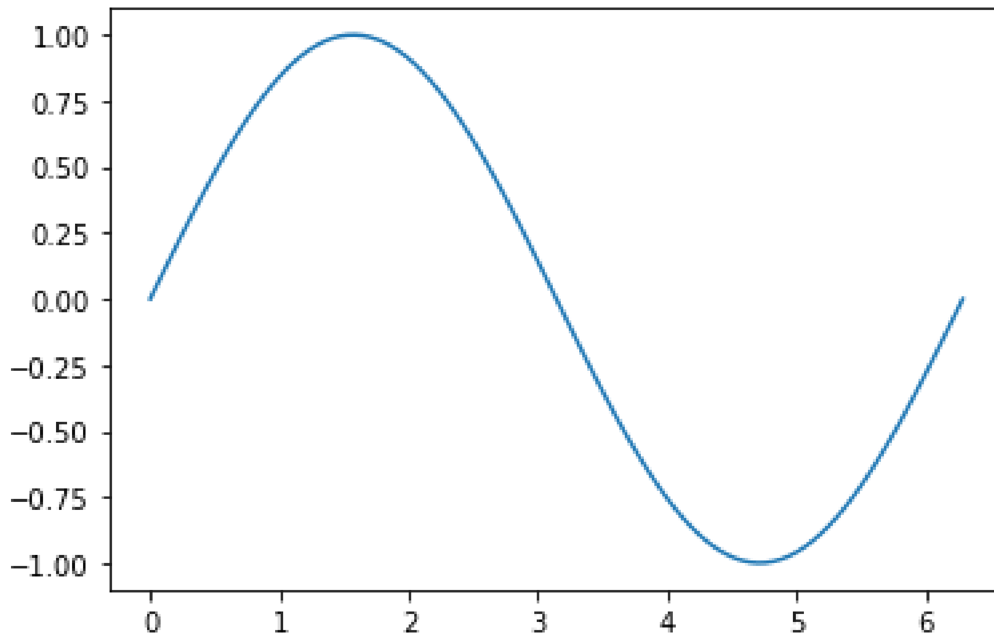
4.1 Simple line plots

Matplotlib is a package that interfaces easily with numpy for the purpose of generating scientific plots and graphics. It should be noted that matplotlib is not the only plotting package commonly used in the Python community, but it is the most popular.

First, let's just generate a simple line graph.

[29]:
```
import matplotlib.pyplot as plt # most common functions are contained in
 ↪"pyplot"

x = np.linspace(0, 2*np.pi, 100) # x-values of points
y = np.sin(x) # y-values of points

plt.plot(x, y) # plot y vs. x
```

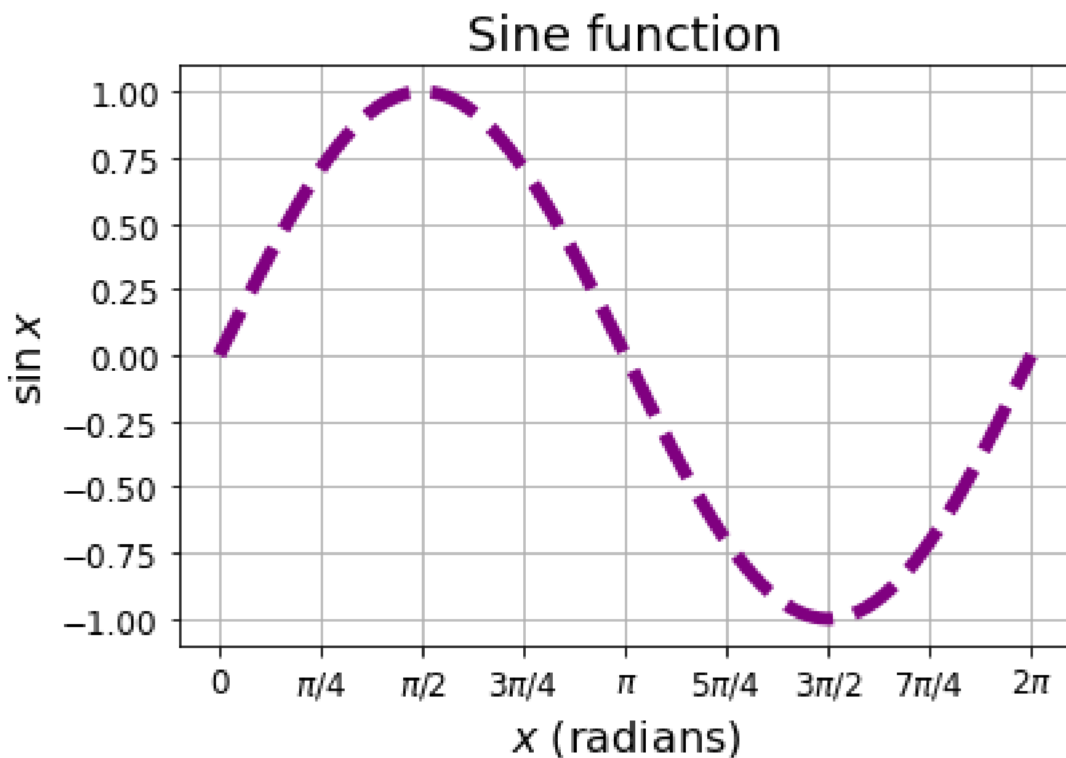
[29]:
```
[<matplotlib.lines.Line2D at 0x1776ed6ffa0>]
```



That was easy but not very aesthetic. Here are some easy steps to make the plot look more professional.

[30]:
```
x = np.linspace(0, 2*np.pi, 500) # x-values of points
y = np.sin(x) # y-values of points

plt.plot(x, y, linestyle='--', linewidth=5, color='purple') # plot y vs. x

title_size = 18
plt.title('Sine function', size=title_size)
plt.grid(True)

# we can use LaTeX in matplotlib if enclosed in $ signs
axes_label_size = 16
plt.ylabel(r'$\sin x$', size=axes_label_size)
plt.xlabel(r'$x$ (radians)', size=axes_label_size)

# put only x ticks in desired places
# and label them with nice LaTeX labels
tick_label_size = 12
tick_locs = np.arange(0, 2*np.pi+np.pi/4, np.pi/4)
tick_labels = ['0', r'$\pi/4$', r'$\pi/2$', r'$3\pi/4$', r'$\pi$', r'$5\pi/4$',␣
↪r'$3\pi/2$', r'$7\pi/4$', r'$2\pi$']
plt.xticks(ticks=tick_locs, labels=tick_labels, size=tick_label_size)
plt.yticks(size=tick_label_size)
```

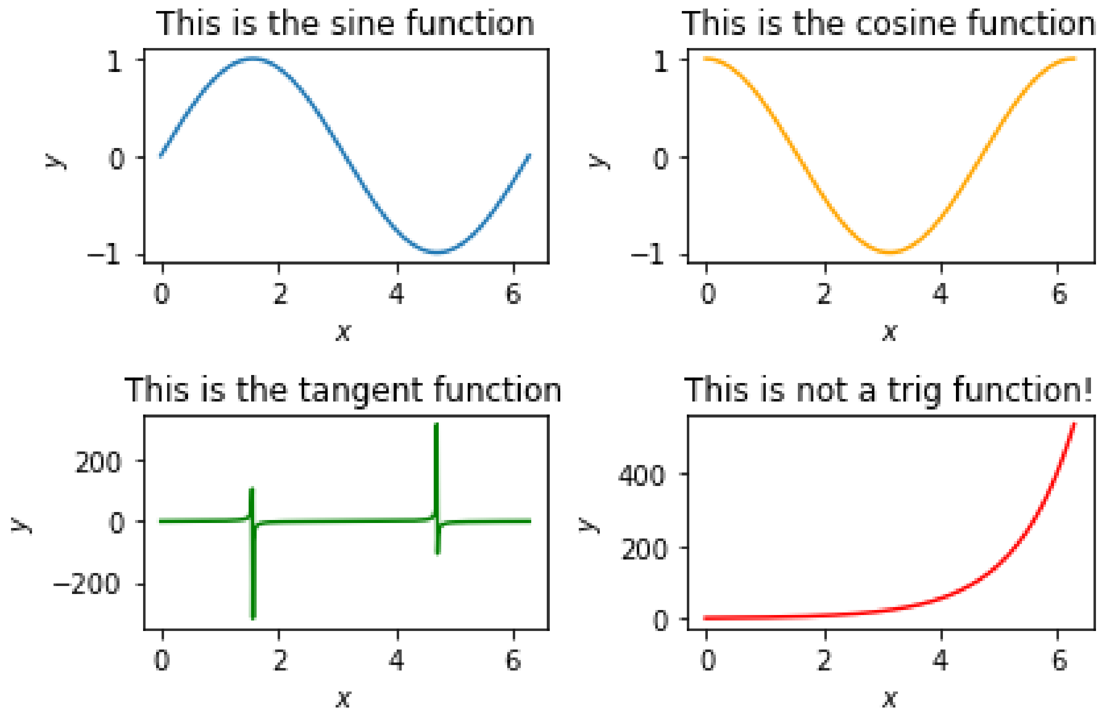
```
[30]: (array([-1.25, -1.  , -0.75, -0.5 , -0.25,  0.  ,  0.25,  0.5 ,  0.75,
         1.  ,  1.25]),
 [Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, ''),
  Text(0, 0, '')])
```



4.2 Subplots

It is also convenient to be able to make a grid of plots to display as subplots. When doing so, we first generate a reference to the figure and its child axes using the subplots command. All plotting is then done using the child axes - often the plotting options when plotting on a subaxis directly have slightly different names than when calling these methods from plt directly. This can lead to significant confusion, so the matplotlib documentation is your friend here.

[31]:
```
fig, axs = plt.subplots(2,2) # this will give us a figure reference, and a 2x2␣
↪array of axes ax

axs[0,0].plot(x, y) # plot the sine function in the top-left sub-axis
axs[0,0].set_title('This is the sine function')
axs[0,0].set_xlabel(r'$x$')
axs[0,0].set_ylabel(r'$y$')

# plot in the top right
y2 = np.cos(x)
axs[0,1].plot(x,y2,c='orange')
axs[0,1].set_title('This is the cosine function')
axs[0,1].set_xlabel(r'$x$')
axs[0,1].set_ylabel(r'$y$')

# plot in the bottom left
y3 = np.tan(x)
axs[1,0].plot(x,y3,c='green')
axs[1,0].set_title('This is the tangent function')
axs[1,0].set_xlabel(r'$x$')
axs[1,0].set_ylabel(r'$y$')

# plot in the bottom right
y3 = np.exp(x)
axs[1,1].plot(x,y3,c='red')
axs[1,1].set_title('This is not a trig function!')
axs[1,1].set_xlabel(r'$x$')
axs[1,1].set_ylabel(r'$y$')

plt.tight_layout() # spaces out the subplots nicely
```



4.3 Heatmaps

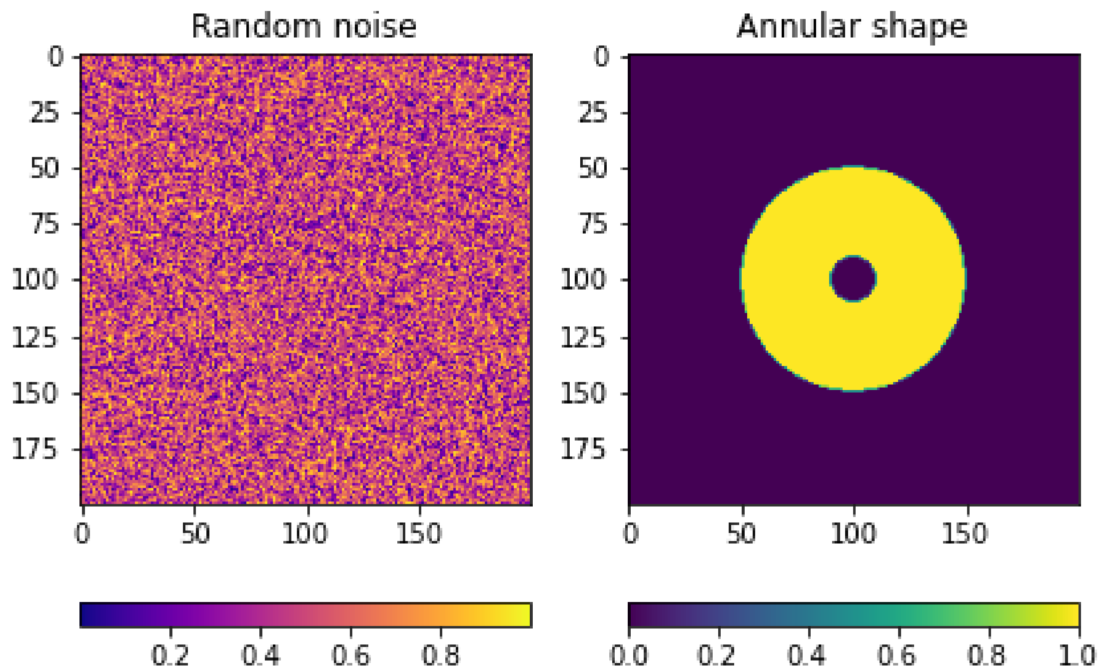
With two-dimensional (i.e., image data) a convenient representation is a heatmap. A 2D array can be displayed as an image using matplotlib's imshow method. This will be useful throughout this course. The colorscheme with which the data is displayed can be controlled. The colormaps available by default in matplotlib are described here: https://matplotlib.org/stable/tutorials/colors/colormaps.html

Different situations call for different colormaps. For example, when dealing with cyclical variables (e.g., such as angles, in which 0 and 360 degrees represent the same point), a cylical colormap should be used.

Below we detail two examples of plotting using a heatmap.

[32]:
```
fig, axs = plt.subplots(1,2)

# first, display random data with some colormap
N, M = 200, 200
random = np.random.rand(200,200)
random_plot = axs[0].imshow(random, cmap='plasma') # plot the random noise and␣
 ↪select jet colormap
axs[0].set_title('Random noise')
# generate a colorbar, have it share the axes with the plot
plt.colorbar(random_plot, ax=axs[0], orientation='horizontal')
```

```
# now, generate an annulus shape
outer_radius = 0.5
inner_radius = 0.1
x, y = np.linspace(-1, 1, M), np.linspace(-1, 1, N)
X, Y = np.meshgrid(x, y)
annulus = np.zeros([N, M])
# find the indices between the inner and outer radii
outer_circle = (X**2+Y**2<=outer_radius**2)
inner_circle = (X**2+Y**2<=inner_radius**2)
annulus[outer_circle] = 1
annulus[inner_circle] = 0

annulus_plot = axs[1].imshow(annulus)
axs[1].set_title('Annular shape')
plt.colorbar(annulus_plot, ax=axs[1], orientation='horizontal')

fig.tight_layout()
```

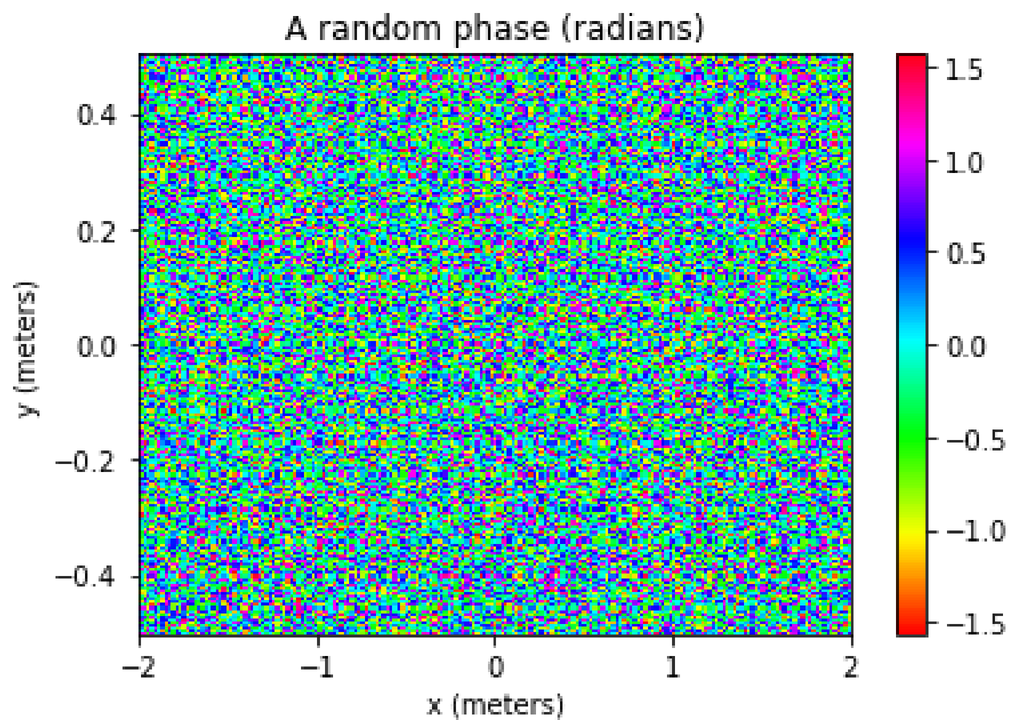


Sometimes we will be showing a heatmap of a region whose x and y axes correspond to distances in physical units. In that case one should use a related function of matplotlib called pcolor. This works as follows. The following example also demonstrates a cyclical quantity (an angle) and a cyclical colormap.

[33]:
```
x, y = np.linspace(-2, 2, 200), np.linspace(-0.5, 0.5, 200)
X, Y = np.meshgrid(x, y)
```

```
data = np.arcsin(2*(np.random.rand(X.shape[0], X.shape[1])-0.5))

plt.pcolor(X, Y, data, cmap = 'hsv')
plt.colorbar()
plt.xlabel('x (meters)')
plt.ylabel('y (meters)')
plt.title('A random phase (radians)')
```

[33]: Text(0.5, 1.0, 'A random phase (radians)')



This way, matplotlib takes care of labeling the x and y axes with the appropriate units.