ES3J1_Lecture_Lab_1

January 17, 2023

1 ES3J1 Advanced Systems and Software Engineering

1.1 Python Lecture/Lab 1: Basics of Python and Jupyter Notebooks

Welcome to **ES3J1 Advanced Systems and Software Engineering**. The purpose of this part of the module is to transfer some of your existing knowledge regarding computational science and engineering using **Matlab** into a new setting: the **Python** language, as used in interactive **Jupyter notebooks**.

Python can also be used in an interactive mode from the command line, or one can write Python scripts (the usual extension is `.py`) that can be run much like JavaScript or Bash scripts, and some Python users prefer to work this way.

In contrast to, say, C++, Python is a non-compiled language and is dynamically typed. The practical consequences of this are that programs/scripts written in Python are generally slower than their C++ counterparts but are also more user-friendly, at least for beginners.

A Jupyter notebook offers an interactive programming environment in which code is broken up into blocks. By default, the code in these blocks is run silently, and so to get text output one needs to use the `print()` command, like this:

[1]:
```
print("Hello World")
import numpy
```

```
Hello World
```

Lab Task. Ensure that you can run a Jupyter notebook either on your own computer or one of the University's computers. Ensure that the commands `import numpy`, `import scipy`, `import datetime` and `import matplotlib` all run successfully, as these Python modules will be needed in the course of the ES3J1 labs and assignments.

We often print a combination of pre-defined strings and some variables, and variables can be formatted and inserted into such strings using the `%` operator and various formatting commands like `%s` for strings, `%f` for floating point numbers, etc. An alternative way of achieving the same results, which some users prefer, is the use of "f-strings".

[2]:
```
from datetime import date
t = date.today()
print("Hello World. Today is %s." % t)
x = 0.123456
print("x is %s." % x)
```

```
print("x is %.3f to 3 decimal places." % x)
### And here is another way to print the same strings
print(f"Hello World. Today is {t}.")
print(f"x is {x}.")
print(f"x is {x:.3} to 3 decimal places.")
```

```
Hello World. Today is 2023-01-17.
x is 0.123456.
x is 0.123 to 3 decimal places.
Hello World. Today is 2023-01-17.
x is 0.123456.
x is 0.123 to 3 decimal places.
```

1.2 Basic data types

Some of the **basic data types** in python include **strings**, **integers**, **Booleans** and **floats**. These can be grouped together into **lists** (and other list-like objects such as **tuples**, **dictionaries**, and **arrays** – but more on those later).

[3]:
```
x = 1
y = 2
print(x + y)
```

```
3
```

Whereas a single equals sign (`=`) is used to assign a value to a variable, a double equals sign (`==`) is used to test for equality. That is, for two variables `a` and `b`, the expression `a == b` is a Boolean variable, either `False` ot `True`. For an even more strict test of equality, use the keyword `is`.

[4]:
```
z = 2.0
print(f"y == z? {y == z}")
print(f"y is z? {y is z}")
```

```
y == z? True
y is z? False
```

The difference between `y == z` and `y is z` above is down to the fact that Python treats integers and floats as being different even if they have the same numerical value, because they are different data types. Another good example of the difference between `==` and `is` comes up when we compare the integers `0` and `1` to the Boolean values `False` and `True`:

[5]:
```
print(f"0 == False? {0 == False}")
print(f"1 == True? {1 == True}")
print(f"0 is False? {0 is False}")
print(f"1 is True? {1 is True}")
print(f"2 == True? {2 == True}") ## NB Python differs from many other languages␣
↪here!
```

```
0 == False? True
1 == True? True
0 is False? False
```

```
1 is True? False
2 == True? False
```

```
<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:3: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:4: SyntaxWarning: "is" with a literal. Did you mean "=="?
/var/folders/m7/93jtp0bn7730ykdtqhp0nl3h0000gn/T/ipykernel_31642/787371666.py:3:
SyntaxWarning: "is" with a literal. Did you mean "=="?
  print(f"0 is False? {0 is False}")
/var/folders/m7/93jtp0bn7730ykdtqhp0nl3h0000gn/T/ipykernel_31642/787371666.py:4:
SyntaxWarning: "is" with a literal. Did you mean "=="?
  print(f"1 is True? {1 is True}")
```

Booleans can be combined using fairly obvious logical connectives like `and`, `or`, `not` etc.

[6]:
```
print((x < y) and not (y is z))
```

```
True
```

Strings consist of zero or more characters enclosed by matching single or double quotation marks. Whether you use single or double quotes is a matter of choice, but they do need to match; to use the outermost style of quotes inside the string, you will need to escape the inner quotes with a backslash, as in the string `s2` below.

[7]:
```
s1 = "The 'quick' brown fox"
s2 = "jumped over the \"lazy\" dog."
s = s1 + " " + s2 ## Let's concatenate the two strings with a space in between␣
 ↪them
print(s) ## and print the result
s = " ".join([s1, s2,]) ## Let's concatenate the two strings in a better way
print(s) ## and print the result
```

```
The 'quick' brown fox jumped over the "lazy" dog.
The 'quick' brown fox jumped over the "lazy" dog.
```

Strings can be "sliced" to yield substrings such as the first character, the first `n` characters, the last character, every other character, etc. The general format is `string[first:last:stride]`, with all three arguments being optional, as the following examples show:

[8]:
```
print(s[0]) ## The first (zeroth) character of the string s
print(s[0:10]) ## The first 10 characters of the string s
print(s[-4]) ## The last character of the string s
print(s[::2]) ## Every other character in s
print(s[1:8:2]) ## Every other character in s, starting with the first, up to␣
 ↪the eighth
```

```
T
The 'quick
d
```

```
Te'uc'bonfxjme vrte"ay o.
h qi
```

One nice feature of strings in Python is that checking for whether or not one string is a substring of another can be done using the user-friendly keyword `in`:

[9]:
```
print(f"'The' in s? {'The' in s}")
print(f"'Then' in s? {'Then' in s}")
```

```
'The' in s? True
'Then' in s? False
```

1.2.1 Basic program flow and functions

Basic program flow. Like most programming languages, Python has the classic structures of branching (using `if` (with an implied "then") and `else`), and `for` and `while` loops.

[10]:
```
for i in range(6):
    ## We will use the % operator:  for integers a and b,
    ## a % b returns the remainder of a divided by b
    if i % 2 == 0:
        print("%s is even" % i)
    else:
        print("%s is odd" % i)

## The following code should chieve the same result, but
## is condidered "less Pythonic" by purists.
i = 0
while i < 6:
    if i % 2 == 0:
        print("%s is even" % i)
    else:
        print("%s is odd" % i)
    i += 1 ## This line is easy to forget and, without it,
    ## the while loop will never terminate!
```

```
0 is even
1 is odd
2 is even
3 is odd
4 is even
5 is odd
0 is even
1 is odd
2 is even
3 is odd
4 is even
5 is odd
```

Indentation. In contrast to languages such as Java and C++, which mark the scope of blocks

of code using braces, and the indentation is a matter of convention and style, in Python the indentation *defines* the scope and is meaningful. It is therefore very important to pay close attention to indentation in Python.

Basics of functions. In Python, functions are declared using the `def` keyword (short for "definition"), followed by the function name and then the input arguments, enclosed in parentheses. When we call the function, the input arguments are again enclosed in parentheses. Any return values are given using the keyword `return` (or, in the case of generators, `yield`).

[11]:
```
def this_function_does_nothing(x):
    pass ## It is good practice to put this here when you define a
    ## function first and intend to come back later to fill it in.

def squared(x):
    return x * x

X = [1, 2, 3, 4]
Y1 = [squared(x) for x in X]
print(Y1)
Y2 = list(map(squared, X)) ## map() returns an iterable, not a list - more on␣
 ↪this later
print(Y2)
```

```
[1, 4, 9, 16]
[1, 4, 9, 16]
```

There is some computational overhead associated to defning a function and keeping it in memory. On some (admittedly rare) occasions, it can be useful to define **anonymous functions** (also known as **lambda expressions**). Here is an example, completely equivalent to the function `squared` defined above.

[12]:
```
X = [1, 2, 3, 4]
Y3 = list(map(lambda x: x * x, X))
print(Y3)
```

```
[1, 4, 9, 16]
```

1.3 Lists and their relatives

There are four or five related data types that provide useful ways to group other objects together. Each is useful in its own way and they must not be confused: * lists * tuples * sets * dictionaries (dicts) * arrays (not actually "basic" – provided by `numpy`)

1.3.1 Lists

To start with, a **list** is an ordered list of objects, each of which can be almost anything you like, and indexed by consecutive integers starting at 0. The `i`th element of a list `L` is `L[i]`. Basic lists are declared using square brackets. Crucially, lists do not behave like vectors that can be added or multiplied be scalars. Try the examples below to see what `L1 + L2` and `2 * L1` are for lists `L1` and `L2`.

[13]:
```
L1 = ["apple", "banana"]
L2 = [1, True]
print(f"L1 + L2 = {L1 + L2}")
print(f"2 * L1 = {2 * L1}")
```

```
L1 + L2 = ['apple', 'banana', 1, True]
2 * L1 = ['apple', 'banana', 'apple', 'banana']
```

Lists can be sliced to extract elements and sublists, just as we did with strings. (Actually, since strings are just lists of characters, their slicability "comes for free".)

[14]:
```
L3 = ["apple", "banana", "cherry", "deadwood", "elm", "fir", "gum", "hazel", ]
print(f"L3[2] = {L3[2]}")
print(f"L3[-1] = {L3[-1]}")
print(f"L3[:3] = {L3[:3]}")
print(f"L3[::2] = {L3[::2]}")
print(f"L3[1:5:2] = {L3[1:5:2]}")
```

```
L3[2] = cherry
L3[-1] = hazel
L3[:3] = ['apple', 'banana', 'cherry']
L3[::2] = ['apple', 'cherry', 'elm', 'gum']
L3[1:5:2] = ['banana', 'deadwood']
```

One of the interesting features of Python is that lists (and other "iterables") can be built using a syntax that is quite close to plain English. The use of `for` loops over an explicit numerical index is possible, but by no means essential. Indeed, not only does the "Pythonic" approach make the source code easier to read, it also offers performance advantages, since in many cases Python will automatically produce the next element of an iterable object when it is needed, rather than generating and storing all elements of a potentially very large list ahead of time. Here is a simple example of list construction using an explicit `for` loop (in two ways) and the "Pythonic" way of doing things:

[15]:
```
X = range(10)
Y1 = [None for i in range(len(X))]
for i in range(len(X)):
    Y1[i] = X[i] + 2
print("Y1 = %s" % Y1)
Y2 = [X[i] + 2 for i in range(len(X))]
print("Y2 = %s" % Y2)
Y3 = [x + 2 for x in X]
print("Y3 = %s" % Y3)
```

```
Y1 = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
Y2 = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
Y3 = [2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Let's see some basic list operations, function definition, and program flow in action by designing a function that will output a list of all the prime numbers up to and including a prescribed upper limit `N`, which we supply as the function's only argument.

At this point we also introduce some good programming practice in the form of `assert`. The keyword `assert` is used to perform "sanity checks" on the code at runtime by checking that something that we expect to be true actually is, e.g. that a variable that should be positive really is, or that a matrix has the right size. If the `assert` statement is `False`, then the program will halt and throw an error; we can write an optional helpful message that can indicate why the assertion failed and help with debugging.

[16]:
```
def primesUpToN(N):
    assert N >= 0, "Input must be a non-negative scalar"
    if N < 2:
        return []
    P = [2,]
    n = 3
    while n <= N:
        is_prime = True
        for p in P:
            if n % p == 0:
                is_prime = False
        if is_prime:
            P.append(n)
        n += 2
    return P
```

[18]:
```
print(primesUpToN(20))
```

```
[2, 3, 5, 7, 11, 13, 17, 19]
```

Lab Task. Convert the above example into a function that calculates the first N primes (as opposed to the primes that are at most N) and implement the divisibility check in a more pythonic way. (Hint: Look up the functionality offered by the `numpy` methods `np.any(X)` and `np.all(X)` when `X` is a list or array containing Boolean `True` or `False` values.

[19]:
```
def firstNprimes(N):
    pass
```

1.3.2 Tuples

A **tuple** is very like a list, except that its elements are **immutable**, i.e. they cannot be changed once set. This can be a useful safety feature if you want to prevent yourself from accidentally overwriting data in some other part of the program. Whereas lists are declared using square brackets, tuples are declared using parentheses.

[20]:
```
X = ["apple", 42]
Y = ("apple", 42)
print("X and Y have the same zeroth element? %s" % (X[0] == Y[0]))
print("X[1] = %s" % X[1])
X[1] = 7 ## This should be possible since list elements can be changed
print("X[1] = %s" % X[1])
Y[1] = 7 ## This should throw an error, since tuples are immutable
```

```
X and Y have the same zeroth element? True
X[1] = 42
X[1] = 7
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[20], line 7
      5 X[1] = 7 ## This should be possible since list elements can be changed
      6 print("X[1] = %s" % X[1])
----> 7 Y[1] = 7 ## This should throw an error, since tuples are immutable

TypeError: 'tuple' object does not support item assignment
```

Tuples declared using parentheses `(...)` have another subtle difference from lists declared using brackets `[...]`: they are **generators** which produce their next element each time they are called, rather then producing them all at once, which can be an important difference and enabling factor in large memory-bound applications. For example, reading a large file or database line-by-line is often better done using a generator than a list comprehension.

[21]:
```
import sys
n = 100
squared_integers_list = [i**2 for i in range(n)]
squared_integers_gen = (i**2 for i in range(n))
print("squared_integers_list is %s" % squared_integers_list)
print("squared_integers_gen is %s" % squared_integers_gen)
print("squared_integers_list has size %s bytes" % sys.
  ↪getsizeof(squared_integers_list))
print("squared_integers_gen has size %s bytes" % sys.
  ↪getsizeof(squared_integers_gen))
```

```
squared_integers_list is [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144,
169, 196, 225, 256, 289, 324, 361, 400, 441, 484, 529, 576, 625, 676, 729, 784,
841, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600, 1681, 1764,
1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704, 2809, 2916, 3025,
3136, 3249, 3364, 3481, 3600, 3721, 3844, 3969, 4096, 4225, 4356, 4489, 4624,
4761, 4900, 5041, 5184, 5329, 5476, 5625, 5776, 5929, 6084, 6241, 6400, 6561,
6724, 6889, 7056, 7225, 7396, 7569, 7744, 7921, 8100, 8281, 8464, 8649, 8836,
9025, 9216, 9409, 9604, 9801]
squared_integers_gen is <generator object <genexpr> at 0x110d7a500>
squared_integers_list has size 920 bytes
squared_integers_gen has size 104 bytes
```

1.3.3 Sets

A **set** corresponds to the mathematical notion of a set: it behaves very much like a list but without ordering and each element can appear only once.

[22]:
```
X = ["apple", "banana", "cherry", "banana"]
Y = set(X)
print(Y)
print(Y[0]) # This line should throw an error because a set is unordered, and␣
 ↪hence has no 0th eleemnt
```

```
{'banana', 'cherry', 'apple'}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[22], line 4
      2 Y = set(X)
      3 print(Y)
----> 4 print(Y[0]) # This line should throw an error because a set is␣
 ↪unordered, and hence has no 0th eleemnt

TypeError: 'set' object is not subscriptable
```

Sets can be tested for equality, containment, etc. in the way that you'd expect:

[23]:
```
Z = {"apple", "cherry"}
print("Is Z a subset of Y? %s" % Z.issubset(Y))
print("Is Y a subset of Z? %s" % Y.issubset(Z))
print("Is Z equal to Y? %s" % (Z == Y))
```

```
Is Z a subset of Y? True
Is Y a subset of Z? False
Is Z equal to Y? False
```

There are also methods for taking unions, intersections, etc.

[24]:
```
Y = {"apple", "banana", "cherry", "banana"}
Z = {"deadwood", "elm", "cherry"}
print(f"union of Y and Z = {Y.union(Z)}")
print(f"intersection of Y and Z = {Y.intersection(Z)}")
print(f"set difference Y and Z = {Y.difference(Z)}")
```

```
union of Y and Z = {'cherry', 'elm', 'apple', 'banana', 'deadwood'}
intersection of Y and Z = {'cherry'}
set difference Y and Z = {'banana', 'apple'}
```

1.3.4 Dictionaries (dicts)

In a **dict** the indices are no longer integers but strings, which we must specify when building the dict. Like sets, dicts have no ordering and are declared using braces. Recall that a set has no ordering, so we cannot extract, say, the 0th element. However, we can extract an element of a dict using its (string-valued) key.

[25]:
```
X = {"name": "Fido", "species": "dog", "age_in_years": 3, "breed": "Alsatian", }
Y = {"name": "Claws", "species": "cat", "age_in_years": 7, }
pets = [X, Y,]
## So X and Y are dicts and pets is a list
for p in pets:
    print(f"{p['name']} is a {p['species']} and is {p['age_in_years']} years␣
 ↪old.")
    ## Annoying point here:  If we had used double quotes around the array keys,
    ## then we would get errors because of the conflict with the double quotes
    ## around the f-string
```

```
Fido is a dog and is 3 years old.
Claws is a cat and is 7 years old.
```

In the above, `"name"`, `"species"` etc. are called **keys** and `"Fido"`, `"cat"` etc. are called **values**. (Thus, lists can be seen as dicts whose keys are integers.) One nice functionality offered by dicts is the ability to retrieve dict elements that might not have been set, and to use and set a default value in the place of unset values (i.e. when then given key does not exist. Remember that elegantly handling unexpected situations, such as an array value not being properly set, is a hallmark of good programming. Observe the difference in the next two blocks between accessing the dict elements directly and accessing them using the `.setdefault()` method.

[26]:
```
print(X["breed"])
print(Y["breed"]) ## This line should throw an error, since we never defined␣
 ↪Y's breed
```

```
Alsatian
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[26], line 2
      1 print(X["breed"])
----> 2 print(Y["breed"]) ## This line should throw an error, since we never␣
 ↪defined Y's breed

KeyError: 'breed'
```

[27]:
```
print(X.setdefault("breed", "breed unknown"))
print(Y.setdefault("breed", "breed unknown"))
```

```
Alsatian
breed unknown
```

Note, though, that as the term `setdefault` suggests, doing this does change `Y` and give it a well-defined value for the `"breed"` key:

[28]:
```
print(Y)
```

```
{'name': 'Claws', 'species': 'cat', 'age_in_years': 7, 'breed': 'breed unknown'}
```

Lab Task. Write functions to peform the following tasks: 1. Given a string `s`, build and return a dictionary of the form `{"a": n_a, "b", n_b, ..., "z": n_z}`, where `n_a` is the number of appearances of the letter `a` (not distinguishing between lower and upper case) in `s`. *Hint:* You may find the the methods `s1.lower()` and `s2.count(s3)` for appropriate strings `s1`, `s2`, `s3` useful. 2. Given a string `s`, return both the most frequent letter in `s` and how many times it appears. *Hint :* Functions can return two or more values simply by separating them using commas, as in `return val1, val2`.

Test your functions on a long string of your choice.

[29]:
```
def letter_count(s):
    pass

def most_frequent_letter(s):
    pass
```

1.3.5 Preview: Arrays in numpy

The `numpy` package offers **arrays**, which are the data structure that represents vectors, matrices, tensors, etc. We will return to arrays and `numpy` later in the module, but for now let's just see that they behave the way that we expect vectors to behave (and the way that lists don't). Let's initialise two 3d-vectors by convering them from lists, and then calculuate their sum, dot product (in two ways), and vector cross product:

[30]:
```
import numpy as np
x = np.array([1.0, 0.0, 0.5])
y = np.array([0.0, 2.0, 1.0])
print("x + y = %s" % (x + y))
print("dot product of x and y (np.dot as 2-argument function) = %s" % np.dot(x,␣
 ↪y))
print("dot product of x and y (np.dot as an array method) = %s" % x.dot(y))
print("cross product of x and y = %s" % np.cross(x, y))
```

```
x + y = [1.  2.  1.5]
dot product of x and y (np.dot as 2-argument function) = 0.5
dot product of x and y (np.dot as an array method) = 0.5
cross product of x and y = [-1. -1.  2.]
```

Under the hood, `numpy` makes use of highly-optimised pre-compiled and optimised routines for linear algebra and the like. The `numpy` implementations of basic tasks like dot products, solution of linear systems, etc. are *much* faster and more stable than anything you can write yourself in pure Python. As a simple example of this, let's calculate a dot product explicitly using a for loop and compare it to what `numpy` offers. The difference is minusclue for small arrays but is quite significant for large ones.

[31]:
```
import numpy as np
from datetime import datetime as dt

n = 100000
```

```
x = np.random.normal(size=(n,))
y = np.random.normal(size=(n,))

t1 = dt.now()
x_dot_y_for_loop = 0.0
for i in range(n):
    x_dot_y_for_loop += x[i] * y[i]
t2 = dt.now()
print(f"x.y the for-loop way = {x_dot_y_for_loop} and takes {(t2 - t1).
 ↪total_seconds()}s.")

t1 = dt.now()
x_dot_y_pythonic = sum(x[i] * y[i] for i in range(n))
t2 = dt.now()
print(f"x.y the pythonic way = {x_dot_y_pythonic} and takes {(t2 - t1).
 ↪total_seconds()}s.")

t1 = dt.now()
x_dot_y_numpy = np.dot(x, y)
t2 = dt.now()
print(f"x.y via numpy        = {x_dot_y_numpy} and takes {(t2 - t1).
 ↪total_seconds()}s.")
```

```
x.y the for-loop way = -40.721106649866485 and takes 0.052689s.
x.y the pythonic way = -40.721106649866485 and takes 0.026252s.
x.y via numpy        = -40.72110664986141 and takes 0.000178s.
```

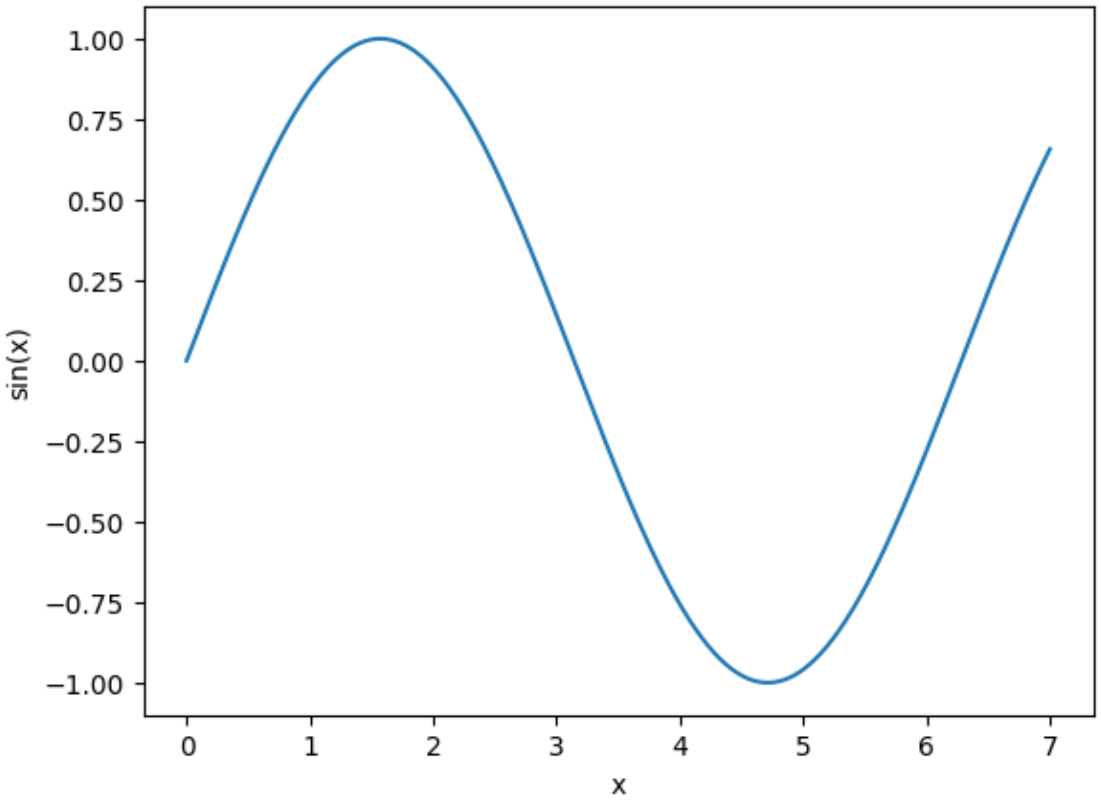
1.4 Basic plotting in matplotlib

There are many data visualisation tools available for Python, but the workhorse and best place to start is `matplotlib`. The `matplotlib` module offers a various visualisation tools for plotting curves, surfaces, and the like. There are other libraries that are better suited to more exotic data types; for example, the `geopandas` module is is well suited to plotting geographical data on maps.

[32]:
```
import matplotlib.pyplot as plt

X = np.arange(0.0, 7.01, 0.01) ## 0.0, 0.01, 0.02, ..., 6.99, 7.0
Y = [np.sin(x) for x in X] ## Actually np.sin(X) would work even better!

fig, ax = plt.subplots()
ax.plot(X, Y)
ax.set_xlabel("x")
ax.set_ylabel("sin(x)");
```

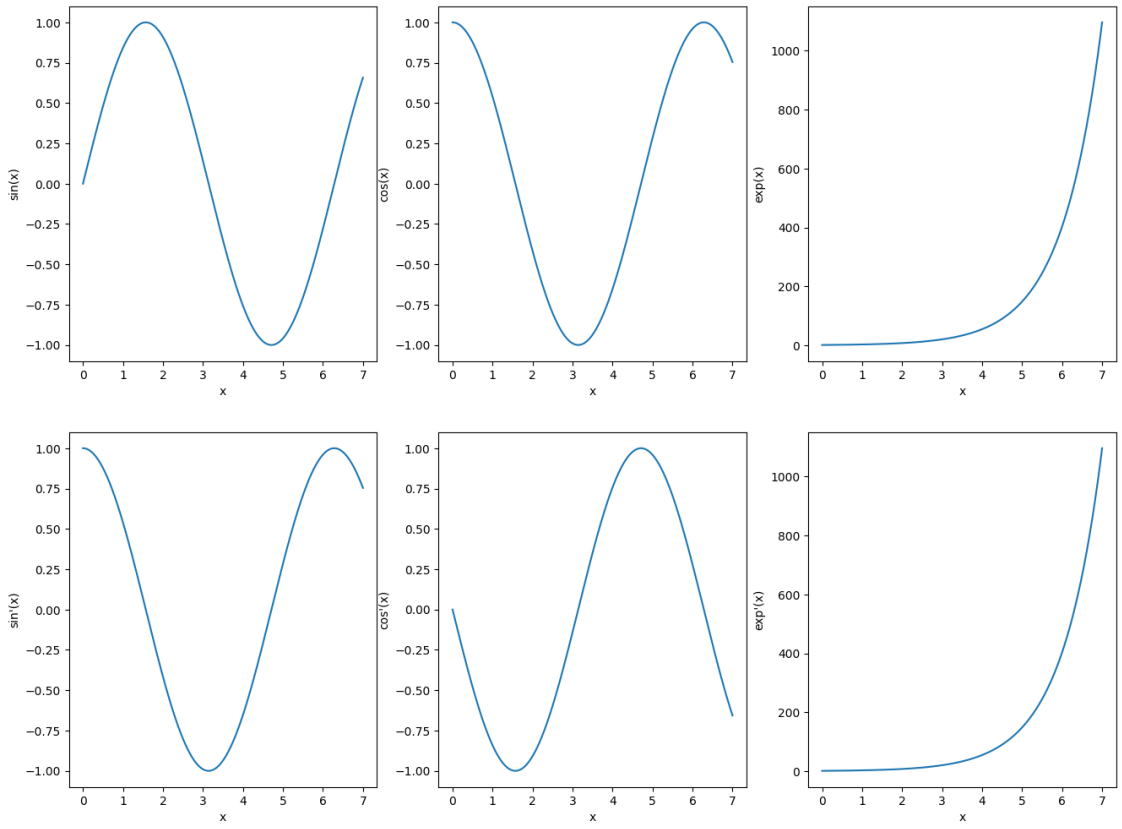


`plt.subplots()` actually creates a rectangular array of axes. If we want to plot multiple functions on different axes, as parts of the same figure, then we do so in the way the next code block shows. Since the total area of the figure has the same default value regardless of the number of subplots, it is often a good idea to change the figure size when using many subplots.

[33]:
```
X = np.arange(0.0, 7.01, 0.01)

fig, ax = plt.subplots(nrows=2, ncols=3, figsize=(16,12))
ax[0,0].plot(X, np.sin(X))
ax[0,0].set_xlabel("x")
ax[0,0].set_ylabel("sin(x)");
ax[0,1].plot(X, np.cos(X))
ax[0,1].set_xlabel("x")
ax[0,1].set_ylabel("cos(x)");
ax[0,2].plot(X, np.exp(X))
ax[0,2].set_xlabel("x")
ax[0,2].set_ylabel("exp(x)");
ax[1,0].plot(X, np.cos(X))
ax[1,0].set_xlabel("x")
ax[1,0].set_ylabel("sin'(x)");
ax[1,1].plot(X, - np.sin(X))
```

```
ax[1,1].set_xlabel("x")
ax[1,1].set_ylabel("cos'(x)");
ax[1,2].plot(X, np.exp(X))
ax[1,2].set_xlabel("x")
ax[1,2].set_ylabel("exp'(x)");
```

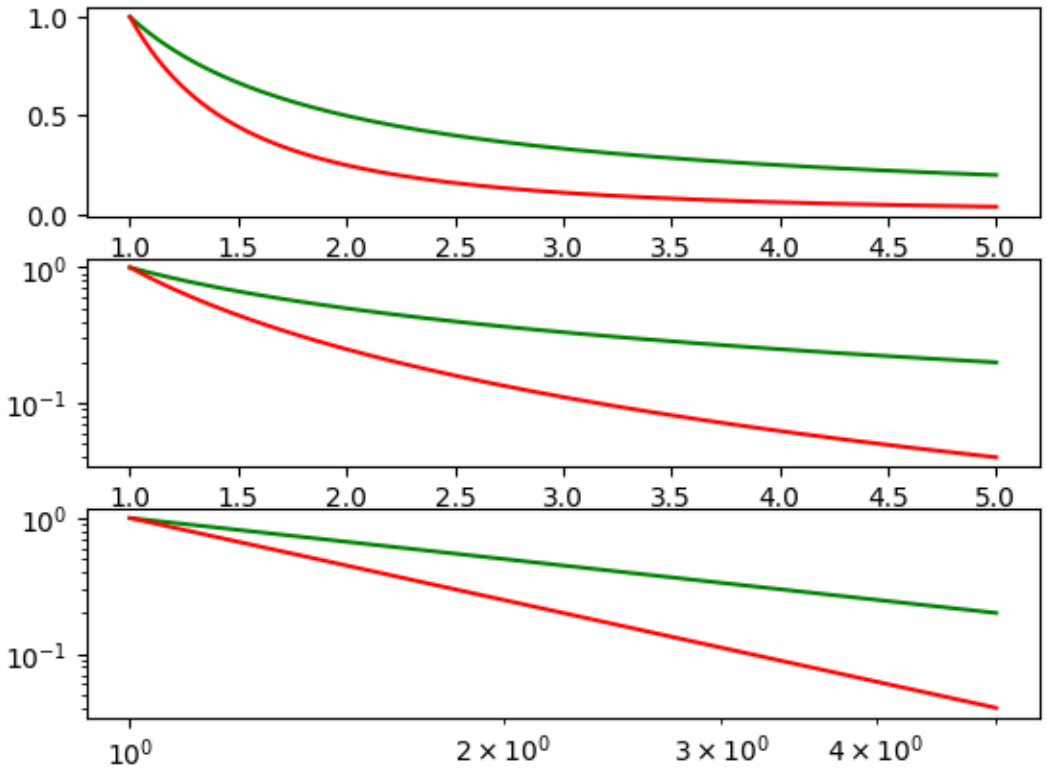


One nice feature of `matplotlib` is that when we need to plot data on, say, logarithmic or semilogarithmic axes, there is no need to create new "transformed" data: instead, we can simply issue a different plotting command. Here is a simple example:

[34]:
```
X = np.arange(1.0, 5.01, 0.01)
Y1 = [x**-1.0 for x in X]
Y2 = [x**-2.0 for x in X]

fig, ax = plt.subplots(3)
ax[0].plot(X, Y1, color="green")
ax[0].plot(X, Y2, color="red")
ax[1].semilogy(X, Y1, color="green")
ax[1].semilogy(X, Y2, color="red")
ax[2].loglog(X, Y1, color="green")
ax[2].loglog(X, Y2, color="red")
```

[34]: [<matplotlib.lines.Line2D at 0x11edf6890>]



Lab Task. The code below will simulate solutions to the famous SIR model (Susceptible, Infected, Recovered) for disease spread, given initial conditions and parameter values.

$$\dot{S} = -\frac{\beta IS}{N} \tag{1}$$

$$\dot{I} = \frac{\beta IS}{N} - \gamma I \tag{2}$$

$$\dot{R} = \gamma I. \tag{3}$$

The function `simulate_SIR` takes as input a `numpy` array of times at which we any approximate values for S, I, and R, a 3-dimensional `numpy` array of their initial values, and values for the constants β and γ; it returns the same array of times and a list of corresponding 3-dimensional (S, I, R) arrays.

Use `matplotlib` to plot the solutions for the initial values $(S, I, R) = (997, 3, 0)$ with $\beta = 0.4$ and $\gamma = 0.04$ on an evenly-spaced grid from $t = 0$ to $t = 100$ with spacing 0.1 in three different ways: 1. three axes within the same figure, one axis for each of S, I, and R; 2. all three of S, I, and R on the same axes; 3. all three of S, I, and R on the same axes, but "stacked" using the `fill_between` method (i.e. the region between 0 and S is coloured in the S colour, the region between S and $S+I$ in the I colour, and the region between $S + I$ and $S + I + R = N$ in the R colour.

[35]:
```
import numpy as np
import matplotlib.pyplot as plt

def simulate_SIR(times, state_0, beta, gamma):
    x = state_0
    assert x.shape == (3,)
    N = np.sum(x)
    X = [x,]
    for i in range(len(times))[1:]:
        dt = times[i] - times[i - 1]
        x = X[-1]
        v = np.array([- beta * x[1] * x[0] / N, beta * x[1] * x[0] / N - gamma␣
 ↪* x[1], gamma * x[1]])
        X.append(x + dt * v)
    return times, X
```

[ ]: