CS181 Code Crash Course

Skyler Wu

January 18, 2022

Contents

1 Welcome!

Welcome to *CS 181: Machine Learning*! CS 181 is primarily taught in Python, so in this Code Crash Course, we will attempt to cover/review almost all the Python you need to succeed in this course. We will assume little to no prior experience with Python for the purposes of this guide. Full disclaimer: this guide is *not* meant to be a comprehensive reference and we strongly recommend that you search up the documentation / additional details to the topics discussed, as necessary.

My teaching philosophy, as you will see in this guide, is to teach by simple example and leave extensive line-by-line comments. There are likely more concise ways to implement some of these examples, but I chose to write out as much as possible (and explicitly state certain default parameters) for the sake of reader understanding.

Logistics-wise, for this course, you are, of course, always free to code in your IDE of choice, but some students may prefer using Jupyter Notebook for ease of debugging and other benefits. For those new to Jupyter Notebook, Jupyter Notebook allows you to run/execute one block/cell of code at a time. This may be helpful when you are prototyping functions or other snippets of code during coding problems on the HW.

To run a code cell/block in a Jupyter Notebook, simply click said cell and then press ctrl+enter (if Windows) or command+return (if Mac). You may also run all cells at once (from top to bottom) by going to "Kernel" -> "Restart & Run All." Note that packages imported in an already-executed cell can be accessed in other cells without reimport. If you are reading the PDF version of this guide, I intentionally refrained from running most cells for sake of saving space + preserving formatting. Please see the Jupyter Notebook version for full outputs.

Please feel free to ask on Ed and/or come to any of the course staff's office hours if you have any questions or need assistance with setting up your computer for the course. We are all here to help!

Some additional resources that you may find helpful are:

1. W3Schools Python Tutorial
2. Numpy Tutorial
3. Another NumPy Tutorial (Video)
4. Matplotlib Tutorial

2 Basics

We begin by introducing some operators commonly used in CS 181 and general computation.

2.0.1 Operators

The operators presented in this section are from vanilla Python and do not require any imports. Later in this guide, you may find that some operators/functions from other packages may have the same use as these vanilla Python operators.

[ ]:
```
# x = 5 (this line of code is commented out and doesn't run). Use "#" for␣
↪individual line comments.

# block comments
'''
This is how you make a block comment in Python (i.e., multiple-line comment).
Note that we use triple quotes -- ''' '''

We use print(stuff) for printing variables/values out.
'''

# addition
print(1 + 1) # 2 (note how we can start a line comment after some code, too!)

# subtraction
print(1 - 1) # 0

# multiplication
print(3 * 5) # 15

# exponentiation (i.e., 2 ** 3 = 2 * 2 * 2 = 8)
print(2 ** 3) # 8

# division
print(7 / 2) # 3.5

# modular division - truncates the fractional part
print(7 // 2) # 3

# modulo arithmetic - 14 mod 5 is congruent to 4 mod 5
print(14 % 5) # 4
```

2.0.2 Assignment (abbreviations)

In computer science, we often want to store our calculations / outputs in variables that we can easily access later. Below, 'x' and 'y' are two variables.

[ ]:
```
# to assign a value to a variable, use the single "=" sign.
x = 1
y = 1

# to change and then update a variable, we have two equivalent methods:

x = x + 1 # this is the universal way
y += 1 # this is the Python abbreviated way. you may also swap "+" with -,*,/,//
 ↪,**, etc.

# let's check the values of x and y after the operations
print(x)
print(y)

# you can also assign multiple variables in one line
a, b = 2+2, 3+3
print(a)
print(b)
```

[ ]:
```
# we can also represent specific values in scientific notation
z = 1e-3
print(z)
```

2.0.3 Comparison

Each of the following comparison operations returns a Boolean value (i.e. True or False). Note that in Python, "True" and "False" are both capitalized!

(alas, quite the source of debugging problems . . .)

[ ]:
```
# let's create some variables
x = 5
y = 7

# checking equality: "==". Note the double-equal-signs.
print(x == 5) # True
print(x == y) # False

# checking inequality: "!="
print(x != 5) # False
print(x != y) # True

# checking greater than/less than
print(x < y) # True
print(x <= 5) # True
print(y > x) # True
print(y >= 5) # True
```

2.0.4 Types and Conversions

Python supports many datatypes – from integers (ints) and decimals (floats), to strings, dictionaries, sets and a whole lot more. Python has some built-in tools for us to check the types of each variable we create/interact with.

[ ]:
```
# suppose we create a variable and assign it a value
x = 100

'''
let's create a string (i.e. a sequence of characters).
In Python, '...' and "..." are both perfectly-valid ways to create strings! We␣
↪will use both arbitrarily.
'''
name = 'Skyler'

# checking types
print(type(x)) # x currently is an 'int' (i.e., integer)
print(type(name)) # name is currently an 'str' (i.e., string)

# converting between types
x_as_string = str(x)
x_as_float = float(x) # we can convert back to an int using int(...)

# notice how 100 became 100.0?
print(x_as_float)

# isinstance tells us whether a variable/object is of a particular class/type --␣
↪returns True or False
print(isinstance(x, str)) # should be False
print(isinstance(x, int)) # should be True
```

2.0.5 If Statements (and abbreviations)

If-statements, also known as conditional statements, tell the computer to do something only if a certain condition has been met. We provide a few examples below.

[ ]:
```
# a standard if-statement
x = 10

if x > 5:
    print("x is greater than 5.")
```

[ ]:
```
# an if-else statement
x = 10

if x < 5:
    print("x is less than 5.")
```

```
else:
    print("x is not less than 5.")
```

[ ]:
```
# an if-elif-else statement
x = 5

if x <5:
    print("x is less than 5.")

# note the syntax here!
elif x == 5:
    print("x is equal to 5.")

# logically, the "else" handles all unspecified cases.
else:
    print("x is greater than 5.")
```

We will soon see that Python is a very reader-friendly language. Keywords like 'and' and 'or' help us create more complex conditional statements.

[ ]:
```
# we can also create more complex conditional statements using 'and'
if (5 < 6) and (6 > 7):
    print("5 is less than 6, and 6 is greater than 7.")
else:
    print("Both expressions are false.")
```

[ ]:
```
# ... we can also use 'or'
if (5 < 6) or (6 > 7):
    print("At least one of the expressions is true.")
else:
    print("Both expressions are false.")
```

[ ]:
```
# we can also abbreviate/shorten our code ...
x = 10

# ... like this (it reads relatively grammatically correctly!)
print("x is less than 11.") if (x < 11)  else print("x is not less than 11.")
```

2.0.6 Writing Functions

Functions are pieces of code that take in some input (or collection of inputs), and does something with said inputs. Canonically, functions will return an output (or collection of outputs), but as we will see later, sometimes functions do not have to return anything.

Let's create a function from scratch that takes in two inputs - 'x' and 'y', and returns the greater of the two. Python and many third-party packages have the max(...) function, but let's build it from scratch for learning purposes.

[ ]:
```python
'''
1. 'def' tells Python we want to define a new custom function.
2. x & y are dummy variables representing our inputs.
3. Note that we do *not* tell Python that x and y must be integers or floats or␣
 ↪some other datatype.
4. we can insert checks that will stop the function if x and y are not int/
 ↪float, but we'll omit that for brevity.
'''

# this tells Python we're creating a new function
def max_from_scratch(x, y):

    # the >= operator only works (as intended) if x and y are both ints/floats
    if x >= y:

        # the 'return' keyword tells Python that this is our final output from␣
 ↪the function (and stop the function from continuing to run)
        return x

    else:

        # we can have multiple 'return' statements in the function, but AT MOST␣
 ↪ONLY ONE should run in any case!
        return y
```

[ ]:
```python
# let's test out our new function - technically, in this case, we can just put␣
 ↪max_from_scratch(100, 50), too.
print(max_from_scratch(x=100, y=50)) # should print 100
```

Python functions can actually return nothing at all, or return multiple values at once. If we want a function to return multiple values at once, we can simply write, for example:

“return x, y, z”

If we want a function to not return anything … well, we can just not write a return statement anywhere in the code.

3 Data Structures

3.0.1 Stock Python Lists

Lists in Python are somewhat similar to arrays in other programming languages. Lists are also indexable (starting from 0). Python lists are somewhat special in the sense that a single list can contain entries of multiple different datatypes, as we will soon show below.

[ ]:
```
# we create a list as such - note the rigid braces (yes, the last entry of the␣
 ↪list is/can be another list!)
list1 = ['Skyler', 'CS181', 3, 4, [100, 200, 300, 400]]

# to access/index into a list, we use the square bracket notation
print(list1[0]) # note that index 0 corresponds to the first entry in the list

# the '-1' index refers to the LAST element in the list! '-2' corresponds to the␣
 ↪2nd-to-last, etc.
print(list1[-1])

# we can add an item to the END of our list as such
list1.append("Add this string to the end of list1")
print(list1)

# we can also remove elements from the list -- alternatively, we could do .
 ↪remove(list1[0])
list1.remove('Skyler')
print(list1)

# to get the length of the list (i.e., the number of entries)
print(len(list1))

# tells us at which index the entry 'CS181' is located (note that 'Skyler' was␣
 ↪deleted)
list1.index('CS181')

# let's create another list
list2 = ['Billy', 'Bob', 'Joe']

# we can combine two lists together using the '+' operator
list_big = list1 + list2
print(list_big)
```

3.0.2 Stock Python Dictionaries

Dictionaries are a data structure relatively unique to Python. At bottom, dictionaries are a set of key-value pairs: each key is mapped to a specific value. Theory-aside, let's see some examples of creating and using dictionaries.

[ ]:

```python
# creating a dictionary from scratch

'''
1. Notice how we use the curly brackets and the colons (indicating key-value␣
 ↪pairs) and the commas (separating entries/pairs)
2. Notice how the values in each key-value pair DO NOT have to be the same type!
3. The dictionary below has values including a string, a list, AND another␣
 ↪dictionary!
4. 'Name', 'Favorite Numbers', and 'Favorite Things' are our keys. The things␣
 ↪they map to are our values.
'''

myDict = {'Name': 'Skyler',
          'Favorite Numbers': [1,2,3,4],
          'Favorite Things': {'Food': 'Noodles',
                              'Color': 'Blue'}}

# accessing a dictionary - let's get my name.
print(myDict['Name'])

# writing an entry to a dictionary - let's add a new key-value pair to myDict
myDict['Favorite University'] = 'Harvard'

# what if we want to access a dictionary inside of a dictionary? Let's get my␣
 ↪favorite food.
print(myDict['Favorite Things']['Food']) # 'Noodles'

# removing element(s) from a dictionary - we remove the entire key-value pair,␣
 ↪based on the key
del myDict['Favorite Numbers']
print(myDict) # just to check that 'Favorite Numbers' is gone

# getting the keys of a dictionary - HOWEVER, 'keys' is a dict_keys object␣
 ↪that's not easy to manipulate
keys = myDict.keys()

# ... instead, we can convert dict_keys to a list
keys = list(myDict.keys())

# getting the values of a dictionary -- similarly, .values() returns a␣
 ↪dict_values object that is not easy to manipulate.
values = list(myDict.values())
```

4 Loops + Control Flow

In general, we use loops when we want to execute a block of code more than once.

4.0.1 For-Loop

We use a for-loop when we want to execute a block of code a *known fixed number* of times.

[ ]:
```
# in Python, we can iterate directly through the elements in a list/other␣
↪iterable datatype
list1 = [2, 4, 6, 'Skyler']

# this tells Python to iterate over every element in list1
for val in list1:

    # note how we directly reference 'val'
    print(val)
```

More commonly, we might want to iterate over a sequence of numbers, and not want to manually create a list. This is when the range(...) function comes in handy.

[ ]:
```
# suppose we want to print out the integers from 4 to 10, inclusive. Written out␣
↪<fully>, we have the following:

'''
1. range(start, stop, step) - this is the format of the range function.
2. First, start=4 (our starting value, inclusive)
3. HOWEVER, stop=11 (our ending value, EXCLUSIVE -- NOT included!)
4. Finally, step=1 (because we are counting by 1s) -- if we wanted to count by␣
↪2s, then step=2
'''
# again, note that I set the second input to be 11 -- the stop position itself␣
↪is NOT included!
for i in range(4, 11, 1):
    print(i)
```

[ ]:
```
# python defaults the start param to 0 and the step param to 1.
# So, if we just wanted 0, 1, 2, 3, 4, we could simply write ...
for i in range(5):
    print(i)
```

4.0.2 Upgrades: enumerate() and zip()

By default, Python iterates over the ENTRIES of a list during a for-loop (if we're iterating over a list). But what if I want to have easy access to the indices as well? Then, the enumerate(...) function comes in handy.

[ ]:
```python
# let's create a list
list1 = ['Apple', 'Banana', 'Can of Beans']

# the enumerate() makes our for-loop iterate over both the indices AND the␣
↪values!
for i, val in enumerate(list1):

    # 'i' represents the indice, and 'val' the actual element in the list.
    print(i, val)
```

But what if I wanted to iterate through two lists at once? The zip(...) function takes in two lists (OR MORE), and returns a list of tuples. We present an illustrative example.

[ ]:
```python
list1 = ['Apple', 'Banana', 'Can of Beans']
list2 = ['Adam Levine', 'Britney Spears', 'Ciara']
list3 = ['A', 'B', 'C']

# the zip() keyword lets us iterate over multiple lists in one pass.
for food, singer, letter in zip(list1, list2, list3):
    print(food, singer, letter)

# but what does zip(...) look like on its own? zip(...) by itself returns a␣
↪not-so-workable object,
# ... so I converted it to a list (a list of tuples to be specfic)
print(list(zip(list1, list2, list3)))
```

4.0.3 While Loop

We use a for-loop when we know how many iterations we want to perform (e.g., iterating through every single value in a list, or every number in a range(...) sequence). But what if we don't know how many iterations we want to perform, but rather want to stop when a certain condition is met? Then we use a while loop.

[ ]:
```python
# suppose I have list of numbers ...
list1 = [1, 3, 5, 6, 9, 10, 14, 16]

# ... and I want to add up these numbers one-at-a-time, and stop at the first␣
↪time this running cum_sum > 20

# cum_sum is our running sum, i is the index we are accessing, starting at index␣
↪0.
cum_sum = 0
i = 0

# keep on adding numbers from our list, if cum_sum < 20
while cum_sum < 20:

    # add the next term to our cum_sum
```

```
    cum_sum += list1[i]

    # increment i
    i += 1

# let's see what our final sum is
print(cum_sum) # should be 24
```

There are generally multiple ways to do the same thing in Python, or any other programming language. We provide an alternate method to solve the above task below.

```
[ ]: # we can also do the same example as above using an 'infinite' while-loop
list1 = [1, 3, 5, 6, 9, 10, 14, 16]
cum_sum = 0
i = 0

# this expression is always True! --> loop goes infinitely ... or does it?
while True:

    # add the next term to our cum_sum
    cum_sum += list1[i]

    # increment i
    i += 1

    # ... we can manually stop the while-loop even if the while condition␣
↪evaluates to True
    if cum_sum > 20:

        # the 'break' keyword tells Python to immediately stop the loop
        '''
        Other useful words:
        1. 'continue' - stop the current iteration, BUT continue with the next␣
↪iteration.
        2. 'pass' - tells Python "there's no code here!" and just read the next␣
↪line of code.
        '''
        break

# let's check our work
print(cum_sum)
```

5 NumPy

Everything we did earlier was in stock Python – i.e., Python right out of the box. However, Python's functionality can be greatly-extended through the importing of 3rd-party packages. One such package that you will use extensively in this class is NumPy, or "Numerical Python." NumPy allows us to easily work with arrays and matrices, and provides us an amazingly extensive collection of mathematical functions to manipulate said arrays and matrices. But first, how do we import a package?

5.0.1 Importing Packages

[1]:
```
'''
1. Typically, we import packages at the beginning of our code, but this seemed␣
 ↪more intuitive organization-wise for this code crash course.
2. After import, we can access methods & classes from the numpy package via np.
 ↪insert_method_name (np is abbrev. for numpy)
'''

# importing 3rd party packages -- here we're telling Python to import and␣
 ↪abbreviate numpy as "np"
import numpy as np
```

5.0.2 Creating Arrays (from existing, from scratch, and randomly)

NumPy allows us to work with arrays and matrices, but we have to create some first, right? Below please find 4 general ways of creating NumPy arrays (oftentimes called np.arrays or nd.arrays for "n-dimensional arrays").

To clarify, we use the words "array," "vector," and "matrix" relatively interchangeably in the context of NumPy.

[ ]:
```
# 1. creating np arrays from existing stock Python structures

## creating np arrays from a list
list1 = [1,2,3,4]

# we can also tell NumPy explicitly what datatype we want the entries to be␣
 ↪using 'dtype'
arr = np.array(list1, dtype=float)
print(arr)

# let's check its type real quick
print(type(arr))

## creating np arrays from a range - we'll create one equivalent to the array␣
 ↪above
arr = np.array(range(1, 5, 1), dtype=float)
print(arr)
```

```
[ ]: # 2. creating np arrays using built-in np functions

     ## an array/matrix of all zeros
     '''
     Remarks:
     1. "matrix," for our purposes, just means multidimensional vector/array (could␣
      ↪be 2D, 3D, etc.)
     2. For np.zeros, np.ones, and np.full, we always have to pass in dimensions as a␣
      ↪TUPLE using (...)!
     3. ^within said tuple, the FIRST TERM is ALWAYS ROWS, and the SECOND TERM is␣
      ↪ALWAYS COLUMNS!
     4. Note that for np.zeros and np.ones, the resultant arrays are ALWAYS defaulted␣
      ↪to be floats!
     '''

     # let's create a 2 row x 3 column matrix of 0s - we can specify dtype if we␣
      ↪want, but default is float, too!
     print(np.zeros((2,3), dtype=float))

     ## let's create an array/matrix of all ones - let's create a 3x3 matrix of all 1s
     print(np.ones((3,3)))

     ## we can also create an array filled with just one value - e.g. a 2x3 array of␣
      ↪all 5s
     print(np.full((2,3), 5, dtype=float)) # note that we specified dtype=float,␣
      ↪because '5' is an int.

     ## we can also create identity matrices (all 0s, except for 1s along main␣
      ↪diagonal): these are ALWAYS SQUARE!
     print(np.eye(3)) # notice how we do NOT specify a tuple of dimensions because␣
      ↪squares have the same no. of rows & cols.
```

```
[ ]: # 3. creating np arrays containing random values

     ## .random() samples values from a Uniform(0, 1) distribution
     print(np.random.random()) # returns a single float
     print(np.random.random(3)) # returns a ONE-DIMENSIONAL array of length 3 filled␣
      ↪with Unif(0,1) draws
     print(np.random.random((3,3))) # returns a 3x3 matrix filled with Unif(0,1) draws


     ## .randn() samples from the standard normal distribution! N(0,1)
     print(np.random.randn()) # returns a single float.
     print(np.random.randn(3)) # returns a ONE-DIMENSIONAL array of length 3 filled␣
      ↪with N(0,1) draws
     print(np.random.randn(3,3)) # returns a 3x3 matrix filled with N(0,1) draws
```

```
'''
Major Notes:
1. It may seem excessive that I seem to have just copy-pasted 3 print-statements␣
 ↪and just swapped out .random() with .randn()
2. HOWEVER, upon closer inspection, one will see that I wrote np.random.
 ↪random((3,3)) and np.random.randn(3,3).
3. This was 100% intentional -- np.random.randn((3,3)) would have returned an␣
 ↪error!
4. Why is this? This is simply a quirk in the implementation.
5. The main point is that one must read the documentation carefully!
'''

## .binomial() in this case samples from the Bin(100, 0.5) distribution.
np.random.binomial(n=100, p=0.5, size=(3,3)) # note that 'size' can also be just␣
 ↪1D if we wanted a 1D array.


## .choice() samples solely from the 1-d array/list 'a' - in this case, from␣
 ↪[0,1,2]
'''
Remarks:
1. 'a' must be a ONE DIMENSIONAL array or just a plain old Python list.
2. 'size' can be an int (for 1D) or a tuple (for a matrix)
3. We can also specify 'replace=False' if we want to sample values from 'a'␣
 ↪WITHOUT replacement.
4. By default, .choice() samples uniformly from 'a' (each value has equal␣
 ↪probability).
5. We can also specify a probability vector p, if needed.
6. Stat connection -- in the default use-case, .choice() is analagous to␣
 ↪DiscreteUniform.
'''
print(np.random.choice(a=[0,1,2], size=(3,3)))
```

In stock Python, we can use the range() function, which is especially helpful for creating sequences for use in for-loops. NumPy provides us with some similar and upgraded functions.

[ ]:
```
# 4. creating sequences

'''
Remarks:
1. Instead of outputting a range object, np.arange outputs an np.array.
2. The .arange() function allows you to specify the dtype.
3. You CAN, when appropriate (i.e. no decimal indexing) replace .range(...) with␣
 ↪.arange(...) in a for-loop.
4. The arange() has some default settings -- see documentation for more concise␣
 ↪code.
```

```
5. Note that like range(), the stop value is NOT INCLUDED in the resultant np.
 ↪array!
'''
## let's try a quick example - I'm writing out the input names for clarity.
print(np.arange(start=0, stop=4, step=0.01, dtype=float))

## .linspace() is a bit similar to .arange() in that it generates a sequence of␣
 ↪values between start & stop
'''
Key Differences:
1. .linspace() takes in a 'num' argument, which tells NumPy HOW MANY values you␣
 ↪want. .arange() takes in stepsize.
2. For example, below, I want 50 values equally spaced between 0 and 4 - NumPy␣
 ↪will calculate the stepsize needed.
3. 'endpoint=True' tells NumPy to include the stopping point as one of the 'num'␣
 ↪values. This is changeable.
'''
print(np.linspace(start=0, stop=4, num=50, endpoint=True))
```

5.0.3 Dimensions, Reshaping, and Recasting

NumPy arrays have specific shapes and even dimensions. We will often have to check the dimensions of our arrays and transform them as necessary for machine learning purposes.

[ ]:
```
# let's create an array from a list
arr1 = np.array([1,2,3,4,5,6])

# let's check its shape using ".shape"
print(arr1.shape) # arr1 is 1D, with shape (6,)

# let's make it 2D - some programs require input to be 2D.
arr1 = np.atleast_2d(arr1)

# let's check its shape again
print(arr1.shape) # now, arr1 is 2D, with shape (1,6) - a row vector.

# we can reshape arr1 to be a 2x3 matrix:
arr1 = arr1.reshape(2,3) # this means 2 rows x 3 columns. The first parameter is␣
 ↪ALWAYS ROWS!

# ... to check.
print(arr1)

# in the special case, we can use .flatten() to make arr1 1D again. This will be␣
 ↪very useful.
arr1 = arr1.flatten()
```

```
# to check.
print(arr1)
```

```
[ ]: # an addendum. what if I didn't really know the dimensions of my array?

# again, let's create an array (pretending we don't know the number of elements)
arr1 = np.array([1,2,3,4,5,6])

# all I know is I want 2 rows.
arr1 = arr1.reshape(2, -1) # the -1 tells numpy to figure out how many columns␣
↪there should be, if 2 rows.

# and, let's check.
print(arr1)
```

5.0.4 Indexing and Slicing

One important NumPy skill is to be able to access specific entries/rows/columns/submatrices of a NumPy array. For ease of visualization, we will stick to a 2D toy array, though the principles extend, of course, to 1D and other multidimensional arrays. Let's create our array that we will use throughout this section.

```
[ ]: '''
For PDF rendering purposes:

arr =
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]
 [21 22 23 24 25]]
'''

# let's use 'arr' as our sample array
arr = np.arange(1, 25+1, 1).reshape(5,5)
print(arr)
```

As we discussed earlier, there are oftentimes multiple ways to do things in Python, and especially in NumPy. Please do not feel discouraged and obligated to know every single way to perform some operation – rather, please use whichever of the following makes the most sense to you.

```
[ ]: '''
A few brief remarks:
1. NumPy numbers its rows from top to bottom (i.e., row 0 is the topmost row)
2. NumPy numbers its columns from left to right (i.e., column 0 is the leftmost␣
↪column)
'''
```

```python
# suppose I want the entry in row 1, column 2 (indexing from 0): I should get 8.␣
 ↪There are two ways to do this.

## first, I can use comma notation
print(arr[1, 2]) # notice how we ALWAYS specify row first, and then column. The␣
 ↪comma separates dimensions.

## second I can use multiple square brackets
'''
Notes:
1. This works because NumPy kind of (unofficially) treats 2D-arrays as a list of␣
 ↪1D row arrays.
2. In a sense, we are first telling NumPy to get us the row at index 1 of said␣
 ↪list: arr[1]
3. Then, we are telling NumPy to get us the entry at index 2 of said row:␣
 ↪arr[1][2]
'''
print(arr[1][2])

# But what if I want the entire row 0?
'''
Notes:
1. The 0 tells NumPy to get us the data at row 0.
2. The colon tells NumPy to return the data for ALL the columns
3. "row 0" + "all the columns" = just get us row 0
'''
print(arr[0,:])

# now, what if I want the entire column 3 (indexed from 0)?
# Similarly, we tell NumPy to get us "all the rows" using the colon, and then␣
 ↪specify column 3
print(arr[:, 3]) # note that arr[:,3] is an ONE DIMENSIONAL ARRAY! We can␣
 ↪reshape it to column or row as needed.

# now, what if I want only the entries [[7,8,9], [12,13,14], [17,18,19]] - i.e.,␣
 ↪the middle 3x3 submatrix?
'''
Let's break down our game plan:
1. Indexing from 0, the middle 3x3 submatrix corresponds to rows 1,2,3
2. Indexing from 0, the middle 3x3 submatrix corresponds to columns 1,2,3, as␣
 ↪well.
3. In NumPy, the notation '1:3' would only get us rows/cols 1 & 2 because 3 (the␣
 ↪ending bound) is EXCLUDED!
4. Thus we need to use 1:4. Let's get it.
'''
```

```
# we can specify the desired rows (first) and columns (second) using the colon␣
↪notation explained above.
# again, the comma "separates dimensions."
print(arr[1:4, 1:4])
```

With the above example, you can generalize the ideas + methods we explored to index/access practically any contiguous pieces of a np.array now!

5.0.5 Mathematical Operations and Broadcasting (Element-Wise)

For this section, we will explore element-wise operations and "broadcasting" (more on that in a moment). To illustrate the relevant principles, we will create two toy 2D arrays to experiment with.

[ ]:
```
# let's create 2 arrays with the same shape (for simplicity) and specify them to␣
↪be floats.
arr1 = np.array([[1,2],[3,4]], dtype=float)
arr2 = np.array([[5,6],[7,8]], dtype=float)

print(arr1)
print(arr2)
```

NumPy has support for standard arithmetic element-wise operations:

[ ]:
```
# we can begin by element-wise adding the corresponding entries in the two arrays
print(arr1 + arr2)

# of course, we can subtract
print(arr1 - arr2)

# we can also ELEMENT-WISE multiply! Note that this is NOT matrix multiplication␣
↪in a linear algebra sense!
print(arr1 * arr2) # we can do division and modular division using / and //␣
↪respectively.

# we can even do element-wise exponentiation!
print(arr2 ** arr1)
```

Now, let us explore "broadcasting." What if I just typed arr * 2? Well, let's try it.

[ ]:
```
# notice how NumPy automatically multiplied element-wise every entry in arr1 by␣
↪2? This is called broadcasting.
print(arr1 * 2)

# of course, we can also broadcast +, -, and other operations -- let's try␣
↪exponentation. Yep, it works!
print(arr1 ** 2)
```

```
# we can also broadcast rows! Let's try adding row 0 of arr2 to arr1
'''
Remarks:
1. Notice how NumPy automatically adds row0 of arr2 to row0 of arr1, and then␣
 ↪adds row0 of arr2 to row1 of arr1?
2. Of course, this broadcasting extends to other operations, as well.
3. Debugging tip: printing out intermediate steps is very helpful in making sure␣
 ↪your broadcasting is successful!
'''
print(arr1 + arr2[0])
```

NumPy also has a variety of fancier element-wise functions that can be directly applied to each individual entry in an np.array.

```
[ ]: # let's try some of these operations out - note that all trig functions are in␣
 ↪RADIANS!
print(np.sin(arr1)) # cos, tan, etc. are defined analagously.

# note that np.log is the NATURAL LOG using base e!
print(np.log(arr1))

# similarly, np.exp refers to taking e to the power of each individual element␣
 ↪in the array!
print(np.exp(arr1))

'''
Two other functions that you can try:
1. np.abs(arr1) - returns the absolute value of each element in an array
2. np.sign(arr2) - for each entry in array, returns -1 if the entry is negative,␣
 ↪0 if equal to 0, and 1 if positive.
'''

# just for kicks
print("Yay!")
```

Another useful tool in NumPy is that we can element-wise evaluate Boolean operations on an array/matrix.

```
[ ]: # recall our 5x5 matrix, arr

'''
Just for the PDF:
arr =
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]
 [21 22 23 24 25]]
```

```
'''

print(arr)
```

```
[ ]: # let's find which elements in 'arr' are equal to 13
print(arr == 13) # this returns a BOOLEAN matrix of True/False values!

'''
Remarks:
1. Again, note that (arr==13) returns a BOOLEAN MATRIX! The only True entry is,␣
 ↪of course, at row 2, column 2
2. In Python, True & 1 are equivalent, and False & 0 are equivalent.
3. Let's see an example.
'''
# let's see whether the above principles hold.
print((arr==13)[2,2] == 1)

# similarly, we can also try other boolean operators (and even compare two␣
 ↪matrices)
# here, we are looking for all entries that are congruent to 0 mod 2 (aka all␣
 ↪even numbers)
print(arr % 2 == 0) # again, this is a BOOLEAN MATRIX!

# what if I want to get all the even entries in this matrix?
print(arr[arr % 2 == 0]) # I can pass this boolean matrix as a 'mask' into arr

# what if I want to get all the indices of all the even entries in this matrix?
# note: each ROW of the resultant matrix corresponds to the indices of an even␣
 ↪entry.
print(np.argwhere(arr % 2 == 0))
```

5.0.6 Matrix Operations: Multiplication, Inverse, Transpose, and Eigenvalues/vectors

Now, we proceed to explore some linear algebra matrix operations. Note that the dimensions/properties on the relevant matrices must be appropriate, or else NumPy will display errors!

```
[ ]: # recall arr, arr1, and arr2 from earlier:
'''
For PDF purposes:

1. arr =
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]
 [21 22 23 24 25]]
```

```
2. arr1 =
[[1. 2.]
 [3. 4.]]

3. arr2 =
[[5. 6.]
 [7. 8.]]
'''

# just to refresh our memory on what these arrays/matrices look like.
print(arr)
print(arr1)
print(arr2)
```

Some commonly-used linear algebra matrix operations include matrix multiplication, taking the transpose of a matrix, finding the inverse/pseudoinverse of a matrix, calculating the trace of a matrix, and finding the eigenvalues and eigenvectors of a square marix.

```
[ ]: # matrix multiplication - make sure the dimensions checkout! For the 2D case, we␣
     ↪have two ways of doing this.
'''
Remarks:
1. np.dot computes the dot product if two 1-dimensional arrays are passed in.
2. np.matmul can also do some higher-dimensional broadcasting tricks (not really␣
 ↪needed in 181)
3. For our purposes, np.dot and np.matmul are similar.
4. Note that both lines of code are performing arr1 x arr2 and NOT arr2 x arr1!␣
 ↪Order matters!
'''

print(np.dot(arr1, arr2))
print(np.matmul(arr1, arr2))

# matrix transpose - again, there are two equivalent ways to this.
print(arr.T)
print(np.matrix.transpose(arr))

# matrix inverse - this only works if the matrix is actually invertible!
print(np.linalg.inv(arr1)) # note that 'arr' is actually singular/not invertible!
 ↪ but arr1 is invertible!

# matrix pseudoinverse
'''
1. Recall that not every square matrix is invertible, but every square matrix␣
 ↪has pseudoinverse.
2. Recall that if X is invertible, then its inverse and its pseudoinverse are␣
 ↪the same!
```

```
3. Note that ALL MATRICES, regardless of dimension, have a pseudoinverse!
'''
print(np.linalg.pinv(arr))

# matrix trace - returns the sum of the entries along the main diagonal.
print(np.matrix.trace(arr))

'''
There are some other useful, but niche functions that are worth mentioning, but␣
↪we will avoid detailed examples for sake of time:
1. np.cumsum - calculates the cumulative sum (i.e., x1, x1+x2, x1+x2+x3, etc.)␣
↪along an axis of a matrices, or just overall.
2. np.linalg.norm - calculates many different types of norms for matrices/
↪vectors.
3. np.around - rounds the elements in a matrix to a specified number of decimal␣
↪points.
4. np.unique - returns all the unique entries in a matrix.
5. numpy.array_equal - checks whether two arrays have the same shape and␣
↪elements.
6. np.nonzero - returns the indices of all the elements in an array that are␣
↪non-zero (useful with Boolean masks)
7. np.linalg.svd - returns the singular value decomposition of any matrix.
'''

# one final thing - NumPy can automatically find eigenvalues and eigenvectors!
'''
Remarks:
1. Note that .eig returns a TUPLE!
2. The first output contains the eigenvalues
3. The second output contains the eigenvectors (indexed corresponding to the␣
↪eigenvalues).
4. Note that for the second output, each COLUMN is an eigenvector!
'''
print(np.linalg.eig(arr))
```

5.0.7 Within-Matrix Operations

In this section, we will explore some useful functions within a matrix. To provide a useful illustrative example, let's use the following toy matrix:

[ ]:
```
arr = np.array([[3, 6, 5],
                [2, 1, 0],
                [5, 9, 4]])
```

Oftentimes, we may want to find the maximum/minimum within a matrix – here, we are treating our matrices as data structures.

[ ]:
```
'''
Note:
1. argmax tells us the indices of the maximum values along each axis (see␣
 ↪documentation)
2. min and argmin are defined analagously.
'''

# this tells us the maximum value in 'arr' OVERALL
print(np.max(arr))

# this tells us the maximum values in each COLUMN of 'arr'
print(np.max(arr, axis=0))

# this tells us the maximum values in each ROW of 'arr'
print(np.max(arr, axis=1))
```

We might also want to find the sum, product, mean, standard deviation, or other numerical summaries of the values in our matrices.

[ ]:
```
'''
Note:
1. np.prod (product), np.mean, np.std (standard deviation) operate analogously,␣
 ↪with the same axis conventions.
'''

# this tells us the sum of all the entries in 'arr'
print(np.sum(arr))

# tells us the sum of the entries in each COLUMN of 'arr'
print(np.sum(arr, axis=0))

# tells us the sum of the entries in each ROW of 'arr'
print(np.sum(arr, axis=1))
```

5.0.8 Sorting and Argsorting

In this section, we will explore sorting within a matrix. To provide a useful illustrative example, let's use the same toy matrix as the previous section.

[ ]:
```
arr = np.array([[3, 6, 5],
                [2, 1, 0],
                [5, 9, 4]])
```

[ ]:
```
# .sort() allows us to sort the values in an array/matrix in ASCENDING ORDER␣
 ↪(left to right) or (top to bottom)

'''
Major Notes:
```

```
1. np.sort(arr) returns a sorted COPY of 'arr'. The matrix 'arr' itself is␣
↪UNCHANGED!
2. If you want to change 'arr', you can do 'arr = np.sort(arr)'.
'''

## let's try it on an 1D array first - yay! It's sorted!
print(np.sort(np.array([2, 3, 1])))

## now, let's try it on a 2D array - 'arr': the values of 'arr' in each ROW are␣
↪now sorted!
print(np.sort(arr))

## note that when we are working with 2D arrays, we can specify which axis along␣
↪which the sorting occurs.
# this sorts the values in each COLUMN of 'arr'
print(np.sort(arr, axis=0))

# this sorts the values in each ROW of 'arr'
print(np.sort(arr, axis=1)) # by inspection, we see that np.sort(arr) defaults␣
↪to axis=1 if unspecified.

'''
Note:
1. np.argsort(arr) returns a matrix of the INDICES of the original 'arr', sorted␣
↪by the VALUES in 'arr'.
2. The same 'axis' settings apply!
'''

# one last thing: specifying axis=None tells NumPy to flatten the matrix into a␣
↪1D vector and sort accordingly.
print(np.sort(arr, axis=None))
```

5.0.9 Various Forms of Stacking

Oftentimes, we may want to join 2 NumPy matrices/arrays together. This is referred to as "stacking." Recall arr1 and arr2 from earlier. Let's use them as our toy examples:

[ ]:
```
'''
For PDF Convenience:

arr1 =
[[1. 2.]
 [3. 4.]]

arr2 =
[[5. 6.]
 [7. 8.]]
```

```
'''

print(arr1)
print(arr2)
```

For this course, you will likely most often use np.hstack and np.vstack. Let's look at their differences below.

```
[ ]: # hstack - combines 2row x 2col arr1 + 2row x 2col arr2 -> 2row x 4col output:␣
      ↪think "laying train tracks"
     print(np.hstack((arr1, arr2)))

     # vstack - combines 2row x 2col arr1 + 2row x 2col arr2 -> 4row x 2col output:␣
      ↪think "building skyscraper"
     print(np.vstack((arr1, arr2)))

     '''
     Remarks:
     1. Note that for hstack and vstack, arr1 and arr2 must have SOME common␣
      ↪dimension, depending on the stack type.
     2. Note that we sent a TUPLE of arrays (arr1, arr2) into hstack and vstack.
     3. We can also send in a TUPLE of however-many-arrays-we-want, if we wanted to.
     4. Note that hstack((arr1, arr2)) != hstack((arr2, arr1)), and same for vstack.␣
      ↪ORDER MATTERS!
     5. There is also an np.stack which ADDS ANOTHER DIMENSION - i.e., two 2D arrays␣
      ↪becomes 1 3D array.
     6. We will omit extensive discussion of np.stack for the purposes of this course.
     7. np.concatenate((arr1, arr2)) is a more generalized alternative to hstack and␣
      ↪vstack.
     8. Specifying axis=1 for .concatenate() is the same as hstack, and axis=0 is the␣
      ↪same as vstack.
     '''

     # let's just see one quick example of np.concatenate - this should be the same␣
      ↪as hstack, because axis=1
     print(np.concatenate((arr1, arr2), axis=1))
```

5.0.10 Copying Arrays (Yes, Necessary!)

Copying arrays is necessary. Believe me, the author of this document has spent many hours in pain because of tricky memory quirks. Please do not follow in his footsteps. We will demonstrate the importance of this section via example.

```
[ ]: # to demonstrate the necessity of this section, let's create a np array
     x = np.array([1,2,3,4])
```

```
# ... now let's attempt to copy it: NOTE THIS IS NOT THE RIGHT WAY! BAD, BAD,␣
 ↪BAD!
y = x

# now, let's alter y - we only want to alter the copy.
y[0] = 10

# ... as expected, y is changed.
print(y)

# ... but, let's check x too. Darn it. It also changed.
print(x)
```

[ ]:
```
# to fix this, we use the .copy() function:

# again, let's create an array
x = np.array([1,2,3,4])

# now, let's copy it -- for real this time.
y = x.copy()

# now, let's alter y - we only want to alter the copy.
y[0] = 10

# let's check the changes.
print(y)
print(x)

# Indeed, only y was changed. Let's celebrate.
print("YAY!")
```

5.0.11 Meshgrid()

This is a bonus topic, but one that will prove very important when visualizing functions of the form f(x,y) that take in 2 inputs.

[ ]:
```
# ***BONUS - meshgrid
'''
Details:
1. Suppose I want to evaluate a function f(x,y) along all xvalues between (0,10)␣
 ↪and all yvalues between (0,10)
2. I can create a meshgrid of xy values on which I can evaluate my function␣
 ↪f(x,y)
'''
# let's try a simple example
x = np.linspace(0, 10, 11) # 11 equally spaced values between 0 and 10 inclusive.
 ↪ This is 1-D
```

```
y = np.linspace(0, 10, 11) # again, 11 equally spaced values between 0 and 10␣
 ↪inclusive. This is 1-D, too.

# Now, we can create our grid using the 1D arrays (which are kind of like␣
 ↪coordinate axes)
xx, yy = np.meshgrid(x, y) # yes, meshgrid returns a TUPLE!

# let's take a look at what it looks like.
'''
Implications:
1. Notice how we can index through each point on the grid and get its (x,y) pair?
 ↪ There we go!
2. xx is 11x11, yy is 11x11
3. we can use plt.contourf(...) to plot f(x,y) -- we'll explain what plt is␣
 ↪later below.
'''
print(xx)
print(yy)
```

5.0.12 Saving/Loading an .npy file

Sometimes, we might want to save our NumPy arrays for future use, or even for use on another computer. Fortunately, NumPy has built-in .save() and .load() functions for us to save our favorite arrays.

[ ]:
```
# let's create an array that I want to save
save_me = np.array([[1,2,3],
                    [4,5,6],
                    [7,8,9]])
```

We can save 'save_me' by doing the following steps:

[ ]:
```
# 1. create a file object (called 'file') that references a directory on our␣
 ↪computer -- in this case, 'myarray.npy'
'''
Notes:
1. 'wb' means 'write-binary' -- essentially, we are telling Python that we are␣
 ↪writing a new file.
2. We use the 'with open(...)' syntax to prevent file corruption. Once we save␣
 ↪the file, the file is closed.
'''

# 1. create a file object (called 'file') that references a directory on our␣
 ↪computer -- in this case, 'myarray.npy'
with open('myarray.npy', 'wb') as file:

    # 2. save the numpy array called 'save_me' to 'file'
```

```
    np.save(file, save_me)
```

We can load 'save_me' from our computer's file system through a similar procedure:

```
[ ]: # 1. create a file object referencing the directory we want - 'myarray.npy', but␣
      ↪this time in read-binary 'rb' mode
     with open('myarray.npy', 'rb') as file:
         save_me_revived = np.load(file)

     # let's check if it worked
     print(save_me_revived)
```

6 Matplotlib

Matplotlib is a time-tested third party package for plotting and creating other visualizations: line plots, scatter plots, histograms, contour plots, etc. – if you can think of it, Matplotlib likely has an extensive implementation. You will make quite a few plots in this class.

Let's begin by importing the package.

6.0.1 Plotting Basics

[2]:
```
# you will use this import ... quite frequently :)
import matplotlib.pyplot as plt
```

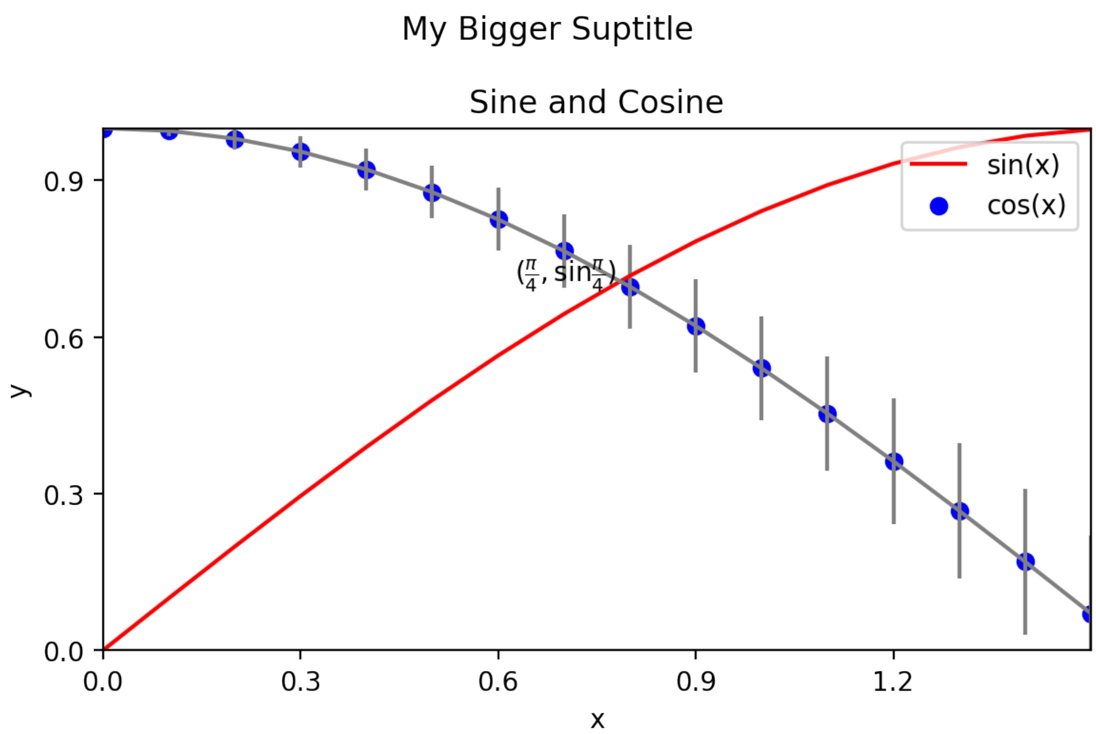
[3]:
```
# let's first generate some mock datasets
x = np.arange(0, 4, 0.1) # go from 0 (inclusive) to 4 (exclusive) in 0.1 step␣
↪increments.

# note that we can broadcast operations on entire arrays!
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.tan(x)
y4 = x ** 2 + 3 * x # x^2 + 3x

# let's just create some error bar lengths for fun
error_bar_lengths = np.arange(0, 0.4, 0.01)
```

Now, let's create a rather-loaded first plot. For pedagogical purposes, we include a lot more features than you will likely have to use in this class.

[4]:
```
# this tells matplotlib that we want to start a figure with width=3, height=2,␣
↪and 200 dpi (resolution)
# ... you could also just do plt.figure() to stick with defaults
plt.figure(figsize=(6, 4), dpi=200)

# this tells matplotlib we want a red LINE graph, with 'x' as our x-variable and␣
↪'y1' as our y-variable.
# the label is useful for when we generate the legend later.
plt.plot(x, y1, color='red', label="sin(x)")

# now, let's add a scatter plot to this SAME figure - you can control dot-size␣
↪with s=<some value>
plt.scatter(x, y2, color='blue', label="cos(x)")

# what if I want error bars on the scatter plot, with length error_bar_lengths?
# there are a lot more settings for this. see documentation. By default, error␣
↪bars are along y-axis.
plt.errorbar(x, y2, error_bar_lengths, color='grey')
```

```
# I want to annotate the point at pi/4: specifically, (pi/4, sin(pi/4))
'''
There's a lot to unpack here:
1. text - this is the text we want add to the plot via annotation. Notice how I␣
↪can type Latex using r"$...$"?
2. xy - this tells us the coordinates of the plot at which we want to place our␣
↪annotation. MUST BE A TUPLE!
3. textcoords - tells matplotlib how we want to offset the text to reduce␣
↪overlap. See documentation.
4. xytext - this tells us how many pts (think pixels) away we want to put the␣
↪text with respect to the xy coords.

(Yes, there's a lot here -- see documentation for details, and please feel free␣
↪to experiment with the code!)
'''

plt.annotate(text=r"$(\frac{\pi}{4}, \sin{\frac{\pi}{4}})$",
             xy=(np.pi/4, np.sin(np.pi/4)),
             textcoords='offset points',
             xytext=(-40,0))

# I want to restrict the view to just x between (0,1.5) and y between (0, 1) -␣
↪this is optional!
plt.xlim(0, 1.5)
plt.ylim(0, 1)

# what if I want ticks every 0.3 increment? - the default is probably fine. this␣
↪is optional!
# these two lines pass np.arrays into xticks(...) and yticks(...)
plt.xticks(np.arange(0, 1.5, 0.3))
plt.yticks(np.arange(0, 1, 0.3))

# note: for some of the above properties, you could also do plt.setp(...) to␣
↪manually alter some settings.

# now let's add our x and y-axis labels + titles
plt.xlabel("x")
plt.ylabel('y')
plt.title("Sine and Cosine")

# we could even add another bigger title - though this will usually be␣
↪unnecessary.
plt.suptitle("My Bigger Suptitle")
```

```
# tells Matplotlib to make a legend out of all the labels we designated. the loc␣
↪argument is OPTIONAL!
plt.legend(loc='upper right')

# I usually call this out of habit -- just to make the plots look a lil nicer
plt.tight_layout()

# this tells plt to save the figure as 'plot.png' in your local directory (i.e.␣
↪same folder as your notebook)
# there are other useful arguments. Check the documentation for additional␣
↪details.
# the facecolor='white' just tells matplotlib to produce solid figures, as␣
↪opposed to transparent.
plt.savefig("plot.png", facecolor="white")

# this line is to properly display the graph - this line MUST come AFTER plt.
↪savefig()!!
# .. after you call plt.show(), matplotlib will put any new operations on a new␣
↪figure.
plt.show()
```



Later on in this course, you may have to plot images (i.e., matrices of pixel values). For sake of length, we will omit an extended discussion of this feature, and simply leave you with the reference to plt.imshow().

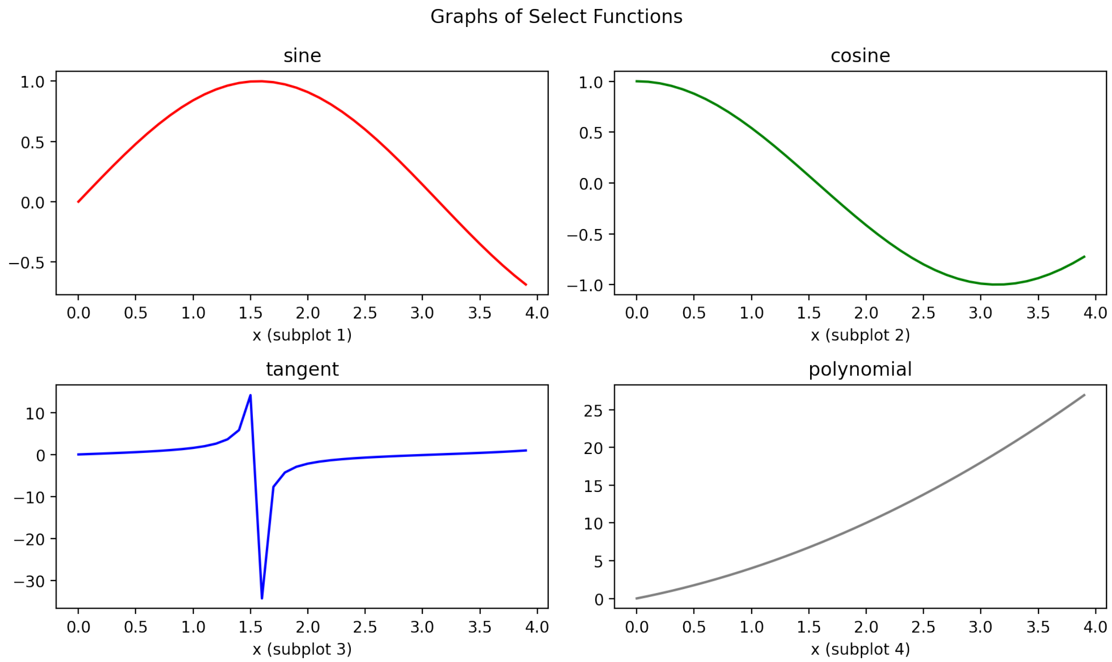
6.0.2 Subplots

We will demonstrate two methods of producing subplots in Matplotlib by plotting the sine, cosine, tangent, and polynomial points that we defined earlier above. Let's just organize our datasets and some descriptive features into lists for ease-of-access in our first method (which uses a for-loop).

[5]:
```
# for our first subplot method, since we are using a for-loop, let's put our 4␣
↪datasets into one list.
data = [y1, y2, y3, y4]

# let's also create a list of labels
labels = ['sine', 'cosine', 'tangent', 'polynomial']

# and a list of colors, because why not?
colors = ['red', 'green', 'blue', 'grey']
```

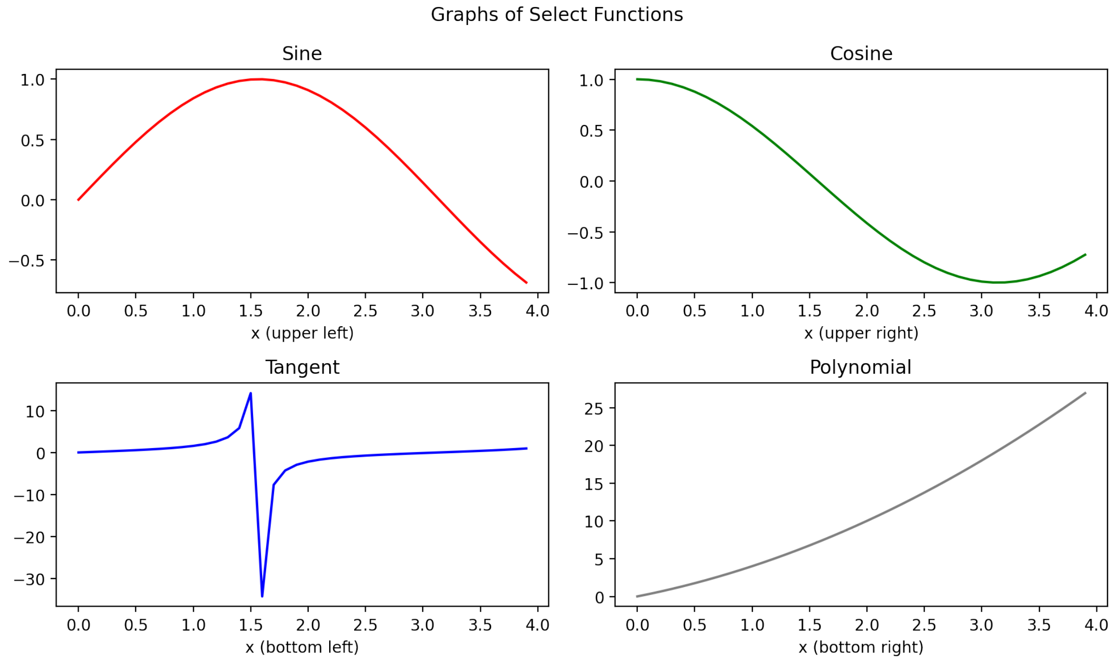
[6]:
```
# first, we tell matplotlib to start a new figure
plt.figure(figsize=(10, 6), dpi=200)

# the odd thing is that in this type of subplot, matplotlib starts counting from␣
↪1, as opposed to 0.
for i in range(1, 5, 1):

    '''
    A few remarks:
    1. Once we call plt.subplot(2,2,i), all subsequent plt.(...) commands only␣
↪affect the current subplot.
    2. By default, matplotlib divides the subplots equally by area.
    3. plt.subplot(2,2,i) tells us we are working with a 2x2 grid of subplots.
    4. i=1 corresponds to the upper left, i=2 to upper right, i=3 to bottom␣
↪left, i=4 to bottom right (etc.)
    '''

    # this tells matplotlib to add a subplot to our figure
    plt.subplot(2, 2, i)

    # let's put the right plot in: the i-1 is necessary because we are starting␣
↪from 1, but lists index from 0.
    plt.plot(x, data[i-1], color=colors[i-1])

    # add in the label + title
    plt.xlabel("x (subplot " + str(i) + ")")
    plt.title(labels[i-1])

# beautify
```

```
plt.suptitle("Graphs of Select Functions") # here, suptitle refers to the␣
↪overall title of all the subplots!
plt.tight_layout()
plt.savefig("subplots1.png", facecolor="white")
plt.show()
```



[7]:
```
# the second way is to use the plt.subplots() function call - let's produce the␣
↪same plot as above
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(10, 6), dpi=200) # this␣
↪creates a grid of 2x2 subplots

# 'ax' is a 2D array of subplots - we directly manipulate the entries in 'ax' to␣
↪produce subplots
# you could more efficiently do the following code in a for-loop, but we'll␣
↪write it out for clarity here.

# upper left
ax[0,0].plot(x, y1, color='red')
ax[0,0].set_xlabel("x (upper left)") # notice how we must use set_xlabel,␣
↪instead of just xlabel because subplots.
ax[0,0].set_title("Sine") # this is the title of JUST the subplot!

'''
A few remarks:
```

```
1. Here, 'ax' is a 2x2 array. The nth row of 'ax' corresponds to the nth row of␣
 ↪subplots
2. We can change xlabel and ylabel within a subplot using set_xlabel,␣
 ↪set_ylabel, etc. (as well as other settings)
3. You could replace 'plot' with 'scatter' or 'bar' or even 'imshow' if you are␣
 ↪working with image data.
'''

# upper right
ax[0,1].plot(x, y2, color='green')
ax[0,1].set_xlabel("x (upper right)")
ax[0,1].set_title("Cosine")

# bottom left
ax[1,0].plot(x, y3, color='blue')
ax[1,0].set_xlabel("x (bottom left)")
ax[1,0].set_title("Tangent")

# bottom right
ax[1,1].plot(x, y4, color='grey')
ax[1,1].set_xlabel("x (bottom right)")
ax[1,1].set_title("Polynomial")

# to beautify - you can use plt to control plot aspects directly, too! now,␣
 ↪we're alterng the entire plot!
plt.suptitle("Graphs of Select Functions") # this is how you control the␣
 ↪suptitle of the entire grid!
plt.tight_layout()
plt.savefig("subplots2.png", facecolor="white")
plt.show()
```



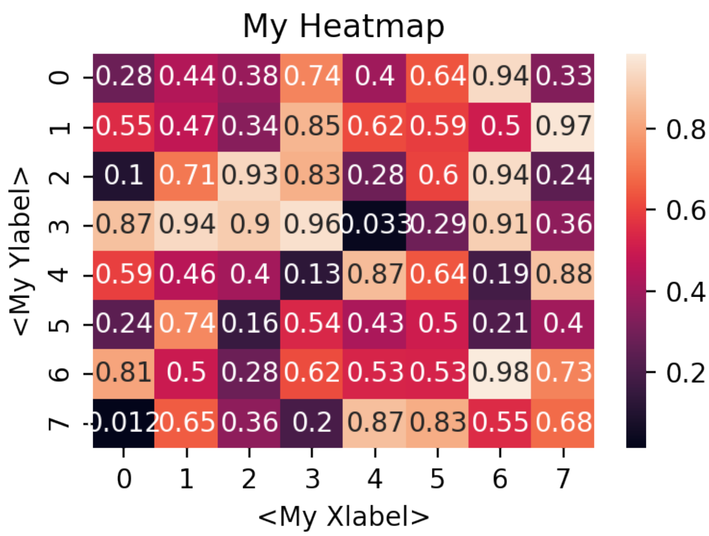
Both methods for producing subplots produce virtually the same results. We recommend you choose whichever one makes more sense to you and/or is easier to implement for a given objective.

6.0.3 Seaborn (for heatmaps)

One question that could come up is, how do we plot a matrix? We can plot a heatmap, where color corresponds to the magnitude of the values in the matrix. Instead of base Matplotlib, we can use another package called "seaborn" that is actually built on top of Matplotlib. We provide an example below.

[8]:
```
# this is the most common abbreviation of 'seaborn'
import seaborn as sns

# first, let's generate a random 8x8 matrix
rand_matrix = np.random.rand(8,8)

# because seaborn is built on-top of matplotlib, we can use matplotlib commands!
plt.figure(figsize=(4,3), dpi=200)

# now, we use sns / seaborn's heatmap function - annot=True just tells sns to
 ↪annotate the actual values, too!
sns.heatmap(rand_matrix, annot=True)

# we can also add titles and labels
plt.xlabel("<My Xlabel>")
```

```
plt.ylabel("<My Ylabel>")
plt.title("My Heatmap")

# beautify, save, and show
plt.tight_layout()
plt.savefig("heatmap.png")
plt.show()
```



7 File Handling (.csv)

In this course, you will, on occasion, have to read data from .csv files. A .csv file, or "comma separated values," can be thought of as a basic Excel spreadsheet, with entries separated by commas (hence the name).

The simplest way I can think of reading a .csv in Python is to use a package called Pandas (coincidentally, my favorite animal, too!). This section *will not* be a full-fledged introduction to Pandas, but just enough for you to be able to import data successfully in this course.

For this section, I have created a toy .csv dataset called 'sec0_code_review_toy_dataset.csv' (representing the price of an imaginary stuffed panda bear over time) with columns 'year' and 'price.' For ease of access, you may download the file from here. Please move this file into the same folder as this Jupyter Notebook.

```
[ ]: # we begin by importing pandas -- this is the most common abbreviation
     import pandas as pd

     # now, we tell pandas to read our .csv as a dataframe (just think of it as a␣
      ↪cool-looking spreadsheet)
     df = pd.read_csv("sec0_code_review_toy_dataset.csv") # df is just a really␣
      ↪common name we use for dataframes

     # let's check that we properly read the file, using the .head() function, which␣
      ↪shows the first couple entries.
     df.head()
```

But how do we make this data play nice with NumPy? Thankfully, Pandas has a 'to_numpy()' function.

```
[ ]: # let's try it! 'arr' just a np.array
     arr = df.to_numpy()

     # let's check, and indeed it works! Now we have our data as a numpy array.
     print(arr)
```

8 Extra Goodies

8.0.1 Select SciPy Functions + How to Read Documentation

SciPy, short for "Scientific Python" is a package that contains some common statistical distributions and mathematical tools. We will demonstrate some select features of SciPy, beginning with its statistical tools.

[ ]:
```
# this syntax tells us to only import specific parts of SciPy.
from scipy.stats import multivariate_normal as mvn # the multivariate normal␣
↪distribution

# for example, scipy allows us to evaluate the multivariate normal distribution␣
↪PDF.

# let's define some arbitrary vector
x = np.array([1.0, 1.5, 2.0])

# let's define a mean vector for our MVN
mu = np.array([0.0, 0.0, 0.0])

# let's make the covariance matrix just the identity matrix
sigma = np.eye(3)

# this evaluates the PDF of the MVN (3-component) distribution with mean mu and␣
↪cov matrix sigma at vector x
# note that we do not call scipy.stats.mvn. Because of how we imported, we can␣
↪directly use mvn.pdf(...)
print(mvn.pdf(x, mu, sigma))
```

Note that scipy.stats has support for many distributions frequently used in classes like Stat 110 and beyond, such as the univariate normal, binomial, and geometric distributions. The documentation and community support for popular packages are very extensive, and I encourage you to dig straight into the documentation and forums!

SciPy also has support for many commonly-used fancy mathematical functions like the sigmoid function.

[ ]:
```
# again, we can elect to import precisely what we need, rather than the entire␣
↪package.
from scipy.special import expit as sigmoid

'''
Fun Fact: sigmoid is used to compress the entire real line to between 0 and 1!
'''

# let's see what sigmoid does
x = np.linspace(start=-4, stop=4, num=100)
y = sigmoid(x)
```

```
plt.figure(dpi=200, figsize=(10,4))
plt.plot(x,y)
plt.xlabel("x-axis")
plt.ylabel("y-axis")
plt.title("Sigmoid Function")
plt.tight_layout()
plt.show()
```

In general, I chose to leave this section relatively-sparse so that you may practice good habits for reading documentation. Below are some suggestions on reading documentation:

1. Read carefully about the inputs and outputs of certain functions and attributes of certain objects! Is there a specific type or dimension of input that you have to reshape your input? Is there an easily-toggleable optional parameter in a certain function that could save you a couple lines of code? The programmers who wrote these packages intentionally included specific features. It's up to you to use them!
2. Many open-source packages like NumPy and SciPy will have direct links to their source code in their documentation pages. If you don't understand what a function is doing or how to read it, look at the raw source code to look under-the-hood!
3. StackOverflow and StackExchange are your best friends (within ethical guidelines)! If you have a question about how a function is used, or why NumPy is throwing dimension errors at you, more likely than not, a couple hundred people have experienced the exact same problem!

8.0.2 List Comprehension

"List Comprehension" is a nifty tool in Python where we can iterate through one list to generate another list in very little code. Yes, this sounds a bit odd, and yes, the code looks a bit odd, too, but let's explore this tool through an example. Suppose I want to generate a NumPy array/matrix where the first row is [1,2,3,4,5,6], the second row is [2,4,6,8,10,12], … so on and so forth, for a grand total of 10 rows. There are many ways to do this, of course, but let's use list comprehension.

Suppose I start with a simple NumPy array of [1,2,3,4,5,6] (presumably from a previous operation). Notice how each row of my intended matrix is just a scalar multiple of [1,2,3,4,5,6]?

[ ]:
```
# this is my starting array
arr = np.array([1,2,3,4,5,6])

# we are going to create a list of np.arrays using list comprehension: pay␣
↪attention to the notation
'''
1. notice how we define 'list_of_arrs' using [...]
2. 'i' is just our indexer/scaler multiple, which goes from 1 to 10, as␣
↪intended, governed by range(...)
3. arr * i is the general element of this list. We are simply varying i.
'''
list_of_arrs = [arr * i for i in range(1, 11, 1)]
```

```
# let's check our results!
print(list_of_arrs)
```

[ ]:
```
# ... unfortunately, list_of_arrs is still a Python list, and not a np array. No␣
↪fear! Let's recast it!
intended_matrix = np.array(list_of_arrs)

# let's check our results. Yay!
print(intended_matrix)
```

8.0.3 Multiple Assignment (Tuples)

Oftentimes, our data will come as tuples – usually in the format of (Class, Features). In Python, we can unpack such data using multiple assignment.

[ ]:
```
# suppose we have a datapoint (i.e., this sample is labeled as class 1, and has␣
↪features [1,2,3,4])
data = (1, np.array([1,2,3,4]))

# multiple assignment - this is just like the enumerate() and zip() we saw␣
↪earlier!
label, features = data # label=1, features=np.array([1,2,3,4])

# we can check if the assignment went through
print(label)
print(features)
```

8.0.4 Defining and Creating Instances of Classes

In this course, you may be asked to write classes of your own. In the context of CS 181, a "class" can be thought of as a blueprint for a tool (e.g. a blueprint for a Logistic Regression classifier). The key point is that we can create individual instances of classes. Sounds convoluted? Let's proceed by example.

To illustrate class construction in Python, let us write a Person class and then define an instance of the Person class called "Skyler." At a high level, we write a class by defining its variables and functions.

[ ]:
```
# we're telling Python that we are now going to describe a class.
class Person:

    '''
    Notes:
    1. This weird '__init__()' function is called the 'constructor.'
    2. Every class has a constructor that is automatically called when we create␣
↪instances of said class.
```

```
    3. In the context of machine learning, you might want to define + initialize␣
↪your weights in the constructor
    4. ^don't worry about what that means for now.
    5. In this context, let's give our Person a name, an age, and a hometown via␣
↪parameters that we pass into the constructor.
    6. Let's also, by default, make every Person we create a CS 181 TF (because␣
↪they are just so cool)
    7. The 'self' keyword is necessary because it tells Python that we want the␣
↪class to be able to talk to itself + store unique instance variables.
    8. Below, our constructor says that in order to create a new Person, we have␣
↪to specify its name, age, hometown, ... and job (see below)
    9. Notice how we prefilled the job parameter? This means that if we do NOT␣
↪explicitly specify job when creating instances of the Person class, they will␣
↪automatically become CS181 TFs
    10. Note that parameters with default settings MUST COME LAST!
    '''

    # notice how we prefilled the job parameter? This
    def __init__(self, name, age, hometown, job="CS 181 TF"):

        # the self.(...) just means that we are going to store the inputs passed␣
↪into the constructor as instance variables
        # instance variables are specific to each instance of the class that we␣
↪create
        self.name = name
        self.age = age
        self.hometown = hometown
        self.job = job

        # let's just have some fun
        print("Person created successfully!")


    # a class has its own methods/functions. Let's create a function so that our␣
↪Person can introduce him/her/themself.
    # we include the 'self' keyword here because we want Person to be able to␣
↪reference self.name, self.age, etc.
    # 'friend_name' is an input to the function -- in context, friend_name␣
↪should be a string.
    # importantly, friend_name can ONLY be accessible within the introduce␣
↪function. It will not carry over to other functions of the class!
    def introduce(self, friend_name):

        # let's implement the greeting
        print("Hello, " + friend_name + ". My name is " + self.name + ", and I␣
↪am from " + self.hometown + ".")
```

```
        # notice how this function does not have 'return'! We could -- but for␣
↪sake of illustration, let's not.


    # let's create another function for fun -- what about a job description?␣
↪Here, we just need 'self'
    def describe_job(self):

        # let's implement the job describing.
        print("I am currently a " + self.job + ".")
```

With our Person class now fully implemented, let's create an instance of the Person class named "Skyler."

[ ]:
```
# notice how I did not specify 'job' - if I want to change the defaults, I can␣
↪add a job="..." argument.
Skyler = Person(name="Skyler", age="<redacted>", hometown="San Diego")

# now, let's ask Skyler to introduce himself
Skyler.introduce("Jack Harlow")

# now, let's ask Skyler to describe his job - notice how nothing is passed in?␣
↪'self' is implicitly passed in.
Skyler.describe_job()
```

8.0.5 Cool Progress Bars (tqdm)

Sometimes, when you are running code, you don't want to just watch a stagnant screen. Why not add some cool-looking progress bars? Also, progress bars can give you a more quantitative sense of how fast your code is running and whether there are any red flags. We use a package called 'tqdm' to enable cool-looking progress bars.

[ ]:
```
from tqdm.notebook import tqdm

# we can wrap tqdm around any iterable to create a cool-looking error bar!
for i in tqdm(range(10000000)):
    pass # this just means "do nothing"
```

8.0.6 Time

Part of studying machine learning is learning to write efficient code. Well, how do we know if our code is efficient? We can time it – and no, put your phone away. Save your phone battery for Instagram. Python has a package called 'time' that can help us out here.

[ ]:
```
import time

# record the start time -- this is in raw seconds since January 1, 1970
```

```python
start_time = time.time()

# do something that should take a bit of time ...
for i in tqdm(range(10000000)):
    pass

# record the end time -- again, this is in raw seconds since January 1, 1970
end_time = time.time()

# calculate time-elapsed (in seconds)
time_elapsed = end_time - start_time

# let's see our results! - notice how the print statement is formatted
print("Start Time: ", start_time)
print("End Time: ", end_time)
print("Time Elapsed: ", time_elapsed)
```

9 Final Remarks

If you are reading this, thank you so much for taking your time to go through this guide / crash course. Note that this guide covered a lot of material at a rather fast pace. If you have any questions or concerns, that is perfectly normal, and we the course staff are here to help you!

Best of luck this semester, and we are glad to have you on-board!