**Python Pointer: isinstance()**

This week we wanted to provide a quick "Pro Tip"; just a small trick you may one day find helpful:

From time-to-time you may wish to check if a variable is of a certain type.  Remember that Python is dynamically-typed so a variable's data type can potentially change between the time it is created and the time it is used.  So let's say, for example, we want to make sure a variable named '**s**' is of type '**str**' before using it in a print statement.  We can use the built-in **isinstance()** function to test this and it will return **True**if the object '**s**' references is of type '**str**' and **False**otherwise.  We will pass two things to the isinstance() function: first is the variable whose type we wish to test and second is the data type we want to test for.  Here's an example:

s = "a string"

is\_string = isinstance(s, "str") # test if s points to a 'str' object

print(is\_string) # prints True

s = 40

is\_string = isinstance(s, "str") # test if s points to a 'str' object

print(is\_string) # prints False

**Joe's Python Pointer: itertools and Generator Expressions**

Since we're talking about iterable objects this week (for loops, while loops, range objects), I wanted to introduce you to a very useful library that is built-in to Python by default: please say hello to [itertools](https://realpython.com/python-itertools/" \t "_blank).  itertools is a library of functions that operate on iterables.  To demonstrate their usefulness, I've prepared [a 4-line example](https://replit.com/@JosephKovba/itertools#main.py) in which we will generate all possible combinations of drawing 5 lottery balls from a pool of 60 (labeled 0-59) and calculate the odds of winning such a lottery.  By using itertools we save from having to use a very nasty nested for loop to generate the combinations ourselves manually.  itertools also provide built-in efficiency measures for making these types of operations run more efficiently too!

In the code you'll also see something on line 13 that looks a lot like a for loop, but slightly different.  This is called a "[generator expression](https://www.python.org/dev/peps/pep-0289/#rationale)".  It iterates over the object returned from the call to itertools.combinations on line 7 and returns a '1' for each and every combination.  Since this is inside of the 'sum' function, the net effect is that we've created a counter that goes up by one for each combination in the set.  The result is a count of all possible outcomes from a lottery of 60 balls from which we select 5!  We then use this to figure out the probability that a single, randomly purchased ticket is a winner.  Something to note about the generator expression, and generators in general: they "generate" values but do not "store" them.  So, a generator will provide us one value at-a-time to work with, but once we're done processing it, the value is NOT stored for later use.  That's perfect for our case because the lottery has over 5.46 million possibilities, so this allows us to count those 5+ million probabilities without having to store them all in memory!

This week we're discussing control structures and conditional statements. As you now know, these are very useful in writing logical, cohesive programs. For those with experience with other languages, you'll notice that there is a type of conditional statement used in many programming languages that we did not mention for Python: the switch statement. In Python, the this is possible in version 3.10, through a mechanism called Pattern Matching.   
  
The match statement can be used to compare a single variable against a number of possible options. Here is an example of a match statement in Python.  
  
i = 2  
match i:  
  case 1:  
    return "1"  
  case 2:  
    return "2"  
  case \_:  
    return "Not 1 or 2!"  
  
When we enter the match statement, the first line of the switch tells us which variable we're using to perform the comparison (match i means we're using the variable i). Then we proceed to check the variable against the case conditions to determine which are true. As you might expect, in the above code, return "2" would get executed. The case \_ is a wildcard condition. Used at the end of the case statements, it will only get executed if no other case statements match.  
  
So why did this *just* get introduced in Python 3.10? Well, you might not be surprised to learn that it has been brought up before: [PEP-3103](https://www.python.org/dev/peps/pep-3103/) . First of all, switch statements are not truly needed by the language, and it was originally deemed that they didn't fit with the style of python code. But what do we mean by not really being needed? Think about how you might create similar functionality to the switch statement above in Python. If you structure if/elif/else chains appropriately, you should be able to construct the equivalent for the pattern matching example above. For example, here is the if/elif/else chain that equates the the above pattern matching.  
  
i = 2  
if (i == 1):  
  print("1")  
elif (i == 2):  
  print("2")  
else:  
  print("Not 1 or 2!")

For more information on pattern matching, check it out [here](https://docs.python.org/3.10/whatsnew/3.10.html) under the Pattern Matching section (PEP 634).

**Joe's Python Pointer: Dynamically-sized Lists**

Python lists are "dynamically-sized" meaning they grow (and shrink) automatically without the developer having to worry about (de)allocating memory.  This is not the case in C, for example, where the programmer must allocate memory (using malloc()) and deallocate memory when finished (using free()).  Trust me, having to keep track of memory manually still gives me nightmares from my undergrad days.  But, how is this accomplished in Python?

Well first, a list in Python is just like everything else in Python: it's an object.  Among other things a list object stores a "pointer" (a memory address) to the beginning of the list as well as the list's size.  Let's pretend we have a list that stores 8 elements beginning at memory location 0 and the list currently has 8 elements in it.  Next, let's insert a 9th element and see what happens.  The list only has space for 8 elements, so Python is going to automatically grow the list for us.  But how does it grow the list, exactly?

A good first guess is that Python grows the list by 1 to accommodate 9 elements.  And, maybe it does the same thing for the 10th, 11th, elements, and so on.  That will work, sure, but it's actually grossly inefficient.  The reason is because of how the expansion of the list occurs.  When a list is grown, Python actually allocates an entirely new segment within memory and copies each of the existing elements into the new list.  So, if your list contains N elements, it requires N copies to move all of the existing elements to over to the new list.  So, using our example above, inserting the 9th element would require copying the 8 existing object references and creating space for the 9th.  Then, adding a 10th element would require copying the existing 9 object references and allocating space for the 10th (and so on).

A better approach is to allocate more memory than is initially needed to avoid having to re-copy all of the existing references each time.  So, how much should the list be grown?  2x, 5x, 10x?  It turns out that if you grow the list too much, this is also inefficient because it wastes a lot of space that might never be used.  While there are different heuristics for growing a list, most languages use a multiplier somewhere between 1-2.  In the CPython interpreter, which we're probably all using for this course, the growth factor for dynamically resizing a list is 112.5%.  So, if you insert the 101st item into a list of size 100, Python will resize the list to hold 112-113 objects.

The logic for shrinking arrays is slightly different, but not too much.  It's a topic unto itself though so we'll skip it for now.  Just be thankful Python abstracts having to think about any of this away!

In addition to the data structures we discussed in class, python has an additional built-in data structure - the frozenset.

Frozensets work very much the same way that sets do. You can run comparisons, copy them determine whether it contains a value. But what you can't do is change the frozenset. Once it is created, it keeps the same elements forever - hence, it is "frozen". A number of the default capabilities from sets are unavailable for frozensets, such as adding, removing, clearing etc.

The most common way to create a frozenset is to use a tuple, which is converted into a frozenset with the built-in keyword "frozenset". For example, see below:

my\_frozenset = frozenset( (1,2,3,4) )

                               this is a tuple ^^^^^^^

So why would you want a frozenset? Doesn't it sound just like a less useful set? Well, there is a very unique feature about frozensets - they are hashable! If you remember from the lecture, sets are not hashable, since they can be altered at will. By eliminating that possibility frozensets are hashable, and can be used as any other hashable data type can - for instance, as a key in a dictionary, or... as a value inside another frozenset.

And that's it! No princesses or magic snowmen required! (Sorry, a joke like that is just too good to *let it go*!) If you want more detail on frozensets, the python documentation for them is here: https://docs.python.org/3/library/stdtypes.html#frozenset

**Alan's Python Pointer - Tarfile**

Alan's Python Pointer - Tarfile

This week we learned about files, and how to read and write from them. One thing that we didn't hit upon is just how large files can get. In modern computing environments, you will frequently see data stored in compressed formats, to save space. This python pointer will discuss one mechanism for file compression: the tarball.

You've probably seen compressed files before. There are many types of compression that you'll see in practice: .zip, .rar, .tar, and many more. So how do you read from, or write to a compressed file in python? One of the most common methods for doing this is the tarfile package in the standard library. "tar"s or "tarball"s are files that contain one or more files compressed within. There are a number of different compression algorithms that might be added on top of tars as well, including gzip, bz2, lzma, etc. In this python pointer, you'll see how to create tarballs and how to extract files from tarballs.

Let's look at an example. Don't worry about the details on the "import" statement below - we'll cover this later in the lecture.  
Compressing files into a tarball.

import tarfile

tar = tarfile.open('my\_tar.tar', 'w')

tar.add('my\_file')

tar.close()

In this case, you can think of "add" as doing roughly the same thing as "write" that you got to see for files.

Extracting data from inside a tarball.

import tarfile

tar = tarfile.open('my\_tar.tar', 'r')

tar.extractall('directory\_to\_extract\_to')

tar.close()

Here, "extractall" will take all of the files that were within the compressed tarball and stick them in the directory that is specified (directory\_to\_extract\_to).

There are lots of useful capabilities of tarfile, including using different compression methods, extracting files by name, and many more. For a full intro to tarfile and its capabilities, check out the documentation on python's website here: <https://docs.python.org/3/library/tarfile.html>

**Joe's Python Pointer: Context Managers**

Something I use at work whenever working with a file, a network/database connection, etc., and something we've noticed students using for the Module 5 Assignment (even though we haven't covered them directly), is called a "context manager".  By the way, we have no problem with you consulting outside sources, our goal is just to make sure you realize what some of these mechanisms are actually doing and why they're so useful.  Also, **please be sure to cite them!** So, if your solution uses the with keyword to open a file this week, congratulations, you're using a "context manager"!

When you use the with keyword to open a file, network sockets, etc., while you're using that resource (the file, for example), you're operating within a "context" of that resource, hence the name.  What you're really doing is working with an object that conforms to the context manager protocol.  This protocol is actually very simple despite the fancy-sounding name, it requires two things for the object being used that the author must adhere to:

1. An \_\_enter\_\_() method: called automatically when you use the with keyword and signifies the start of a context (the use of the resource)
2. An \_\_exit\_\_() method: called automatically when the execution of the program leaves the with block and signifies the end of a context (the user will no longer be used going forward)

So what are the benefits of starting a context manager using the with keyword?  There are a few:

1. They make resource management much easier and protect against resource leaks.  We mentioned in the lecture that it's a best-practice to close a resource when you're done using it, but what happens if you forget?  If you open too many files or network connections, for example, you could end up denial-of-servicing your machine!  When you leave a with block, the resource you opened using the with statement is automatically closed for you without having to close it yourself, sweet!
2. try-catch statements, which can be used to detect errors when using a resource (and we'll see in Module 10), are useful but they require you to know what kinds of exceptions to detect based on the resource you're working with.  Using context managers, you can simply rely on the internal logic of the object to do all of the error handling for you which simplifies your code and reduces the effort required to write a resilient program.

So, if you decide to take the approach of opening a file by doing: with open('"points.txt", "r") as f: to read in the parameters and data points this week, hopefully this Python Pointer has given you a bit better idea of what the open() function is doing behind the scenes!

**Joe & Alan's Python Pointer: Lambda Functions**

In conjunction with this week's topic of functions, we'd like to introduce you to a special type of function: a lambda function. Lambda functions are python's implementation of Lambda Calculus ( https://en.wikipedia.org/wiki/Lambda\_calculus ). Also called anonymous functions, they are special functions that are created without a name, and are useful in scenarios in which you wish to create a function that you only ever want to execute in one location. Let's first take a look at the structure of a lambda function.

lambda x: x\*\*2

1. "lambda" - The first word on the line is "lambda", a special reserved word that python used to declare a lambda function. It is somewhat similar to the keyword "def", in that it is specifically used to initiate the beginning of a function definition.

2. "x" - The next section contains the parameters that will be passed to the function. In this case, we only have one parameter, "x". You could just as easily have multiple parameters, however, with comma separation: e.g. lambda x, y: x + y

3. ":" - The colon indicates the end of the parameters and the beginning of the body of the function.

4. "x\*\*2" - The body of the function. This is where the actual logic for the function will be implemented.

And that's basically it. You can immediately invoke the function: (lambda x: x\*\*2)(2) . This will create the lambda function, and immediately invoke it with the parameter 2. Note the parentheses around the function here, which are needed to syntactically indicate where the function ends. You can also assign it to a variable as a function pointer: functionPointer = lambda x: x + 1 .

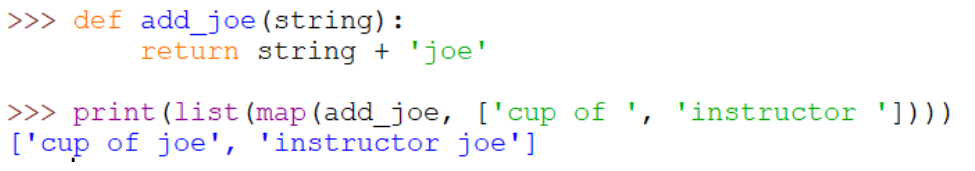
There are, however, a couple of other interesting characteristics that you should know about lambda functions. Lambda functions can ONLY have 1 expression, and must have ZERO statements. Any statements within the lambda will raise a SyntaxError. This includes things like return and raise, but also more subtle statements, such as assignment (=). Also, type hinting/annotation is not allowed inside of lambda functions (how would python be able to tell the difference between type hinting and the end of the lambda parameters?).

Lambda functions may seem a bit niche - and they are. They can be quite convenient in certain scenarios, and I find that I use them most frequently when using map(), reduce(), and filter() functions (which are a bit beyond the scope of this course). A full dive into lambdas can be found here: https://realpython.com/python-lambda/ if you're interested in learning more. It goes much further into the detail for lambdas and their uses than we've gone into here, but also touches on constructs that are far beyond the scope of the course, so be forewarned.

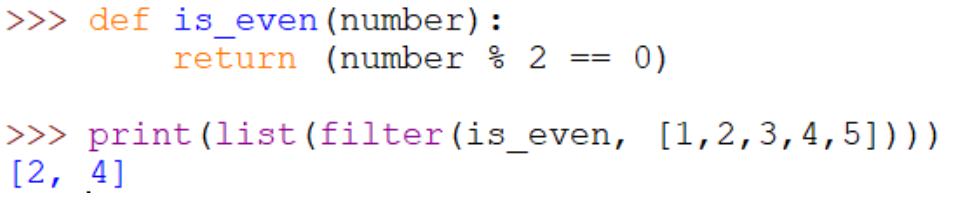
**Advanced Python Pointer: Map/Filter/Reduce**

Now that you’ve learned about lambda functions, let’s take a look at another functional programming concept: map/reduce (https://en.wikipedia.org/wiki/MapReduce). An early mechanism for performing Big Data calculations, map/reduce programming concepts have started to fall out of favor, though there are still modern applications of the logic used in Big Data scenarios today (such as Apache Spark). Furthermore, map/reduce techniques can be used to simplify code and to express certain operations in much cleaner fashions. The 3 main pillars of map/reduce in python are the builtin functions map(), filter(), and reduce().

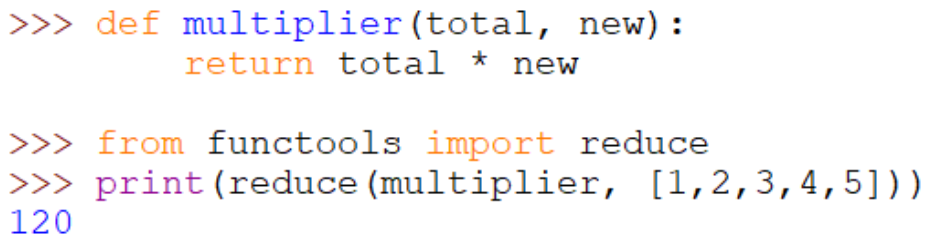
map(function, list) : takes 2 arguments, a function and a list. It then proceeds to perform the function across all the elements of the list. The return value of the function represents a transformed version of the original element. map() returns a map object, that can easily be turned back into another list with the list() function. In a nutshell, map() is a way of performing some kind of transform on all the items in a list. The total number of elements in the new list should be equal to the total number of elements in the original list.



filter(function, list) : like map, takes 2 arguments, a function and list. However, while map is designed to transform all the elements in a list, filter() is designed to winnow down the existing elements in the list. Rather than returning a transformed copy of an element, the filter() function should return a boolean, determining whether the element should be included in the new list. So the total number of elements in the new list after a filter may be less than the total number of elements in the original list. Similar to map, you will need to use the list() function to transform your filter() object.



reduce(function, list, initial=None) : takes 2 required arguments and 1 optional argument. reduce() works differently than map() and filter() in that the end result for a reduce() function is not a list. Reduce()’s job is to convert a list into a singleton. In both map() and filter(), the inner function that was being invoked only had 1 argument, the element that was being processed. Reduce()’s inner function, however, requires 2 arguments, the running total value and the new element in the list. The return value at the end of reduce is the final outcome. Optionally, you can add in initial value for the first iteration as well. One more thing to note about reduce, is that it is no longer a builtin keyword in python. Instead, you must import it from the functools module (don’t worry about the details on this – we'll cover them after the midterm – for now, just know the “from … import ...” line is required).



For more information and examples on map, filter and reduce, check out the writeup here: <https://stackabuse.com/map-filter-and-reduce-in-python-with-examples/>

**Joe & Alan's Python Pointer: Generators & the 'yield' Keyword**

Attached Files:

* [[File](https://blackboard.jhu.edu/bbcswebdav/pid-11175694-dt-content-rid-112186011_2/xid-112186011_2) JHU\_Python\_Module9\_LectureNotes\_OOP2\_PP\_Generators.pdf](https://blackboard.jhu.edu/bbcswebdav/pid-11175694-dt-content-rid-112186011_2/xid-112186011_2) (110.95 KB)

In this week's PowerPoint slides contained in the Video Lectures section you'll notice slides 18-21 contain information on generators and the yield keyword.  These two things, combined together, allow for the efficient processing of large datasets we may not necessarily want to store in memory.  Generators produce elements one-at-a-time but DO NOT store them in RAM the same way a function does.  Instead, we use the yield keyword to "return" the values the same way we would using an iterator.  If you're interested in learning more about these concepts, please check out the slides attached here!

**Joe & Alan's Python Pointer: pylint**

Attached Files:

* [[File](https://blackboard.jhu.edu/bbcswebdav/pid-11175696-dt-content-rid-112186031_2/xid-112186031_2) Module 10 Python Pointer.pdf](https://blackboard.jhu.edu/bbcswebdav/pid-11175696-dt-content-rid-112186031_2/xid-112186031_2) (316.796 KB)

This module's python pointer covers the pylint tool! Please see the attached PDF.

### Module 11 Python Pointer: Google Colab

Hello everyone,

For this week's python pointer, we'd like to introduce you to Google Colab. Colab is Google's version of Jupyter. Jupyter is an open source project and requires the user to host their own server (though this is not terribly difficult). Colab, on the other hand, runs on Google's resources, though they are not always reliable. Colab runs ipynb files (also known as Jupyter Notebooks). The interface is very similar to Jupyter and will feel very familiar to you after viewing the Jupyter slides from this lecture.

In traditional Google style, your Google account will be used to authenticate and log you into Colab. In addition it is integrated into other Google tools including Drive. Also, Jupyter access for multiple participants can be difficult to manage, when hosted servers are hard to establish. Colab eliminates this problem, and allows you to collaborate more easily (hence the name). Also, Google will allow users access to special kernels on an as-available basis. While this might not seem too important, the main point is that you can hook "workhorse" backends to it, including GPU rigs as well as your own computer. This can greatly facilitate machine learning processes.

To access Google Colab, you can go here: [https://colab.research.google.com](https://colab.research.google.com/) and for more information on Colab, check out their FAQ here: <https://research.google.com/colaboratory/faq.html>

**Module 12 Python Pointer: Drawing Graphs**

One thing that you will probably notice throughout this course is that we are don't introduce many visual or GUI elements into this course. Computer graphics and rendering are a very large subgenre of computer science and are surprisingly tricky. Production grade graph rendering takes a lot of effort and time. Fortunately, however, rendering graphs from networkx for simple graphs and prototypes is actually really easy, and is the subject of this week's python pointer.

You may remember matplotlib from the extra (optional) section of the kmeans assignment. This was how we rendered out the clusters. Networkx has done some great work in integrating with matplotlib, so simple rendering is just that - simple! Note that as always, you will need to import matplotlib to be able to use its functionality. Lets take a look:

import networkx as nx

import matplotlib.pyplot as plt

graph = nx.dodecahedral\_graph()

nx.draw(graph)

plt.show()

Graph rendering can occur in all kinds of different layouts and mechanisms. NetworkX with matplotlib expose a number of different methods that you can call to render the graph in different ways.

Circular Layout: nx.draw\_circular()

Random Layout: nx.draw\_random()

Spectral Layout: nx.draw\_spectral()

Spring Layout: nx.draw\_spring()

Shell Layout: nx.draw\_shell()

Graphviz Layout: nx.draw\_graphviz()

You can also specify many more additional configurations and settings by using the nx.draw\_networkx method.

nx.draw\_networkx(graph, with\_labels=True, node\_size=200, node\_color='b', edge\_color='g')

A full description on drawing with networkx can be found here: <https://networkx.org/documentation/networkx-1.10/reference/drawing.html>

**Module 13 Python Pointer: Thread Pools**

This week we've learned about threads and how they can be used to perform concurrent operations. But one thing you might have noticed is that there is a lot of boilerplate\* code needed to actually run thread, what with the creating, the starting, and the joining of those threads. Wouldn't it be nice if it was easier and cleaner to handle that, so that you just feed in the actual logic you want performed concurrently? Enter Thread Pools.

Thread pools have a pretty simple concept. Imagine you had a "pool" of available resources you can draw from. For example, let's imagine you have 10 hammers in your shed. Your buddy comes over and needs to borrow a hammer, so he gets one out of your shed and starts his repair project, so you have 9 hammers left. Maybe another friend comes and uses another hammer, leaving you 8. The hammers here are the available resources in your pool. Once your friends are done using the hammers, they will put them back in the shed and those hammers will be available for someone else to use. If 11 friends attempt to get hammers from your shed, then the 11th friend will have to wait until one of the hammers is returned.

Thread pools work in the same way, except instead of hammers, you have threads. If your thread pool is initialized to 10 threads, then you can easily assign tasks (functions) to those threads to execute, and once the task (function) completes, you recoup the thread resources and ready them for another task.

Below, you'll see the use of thread pools from the concurrent.futures module. Note that we also use a context manager for this example, which was covered back in Module 5's python pointer.

import time

from concurrent.futures import ThreadPoolExecutor

def wait\_and\_print(duration):

    time.sleep(duration)

    print(f'Waiting for {duration}s complete')

t1 = time.time()

with ThreadPoolExecutor(max\_workers=5) as executor:

    for i in range(1,6):

        executor.submit(wait\_and\_print, i)

    print(f'Finished assigning tasks.')

t2 = time.time()

print(f'Total elapsed {t2-t1}s')

Notice that we didn't have to go through all 3 steps for each task we needed completed, we simply submitted the task to the thread pool. You should be able to see that the threads are in fact running concurrently. Now, thread pools are great for bundles of tasks that need to be completed concurrently, but lack some of the fine grained control of executing threads directly. So always consider if thread pools are the right choice for your application before you choose to use them.

The documentation for ThreadPoolExecutors is here if you want to do your own research: https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ThreadPoolExecutor .

\*Boilerplate code is repetitious code that could debatably be removed if different language or library design choices were made. ( https://en.wikipedia.org/wiki/Boilerplate\_code )