# The Zen of Python

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

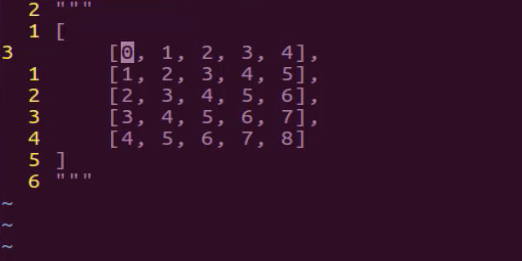
Namespaces are one honking great idea -- let's do more of those!

# Manually Build an Incrementing Matrix Function in Python

In this coding exercise, we're going to walk through how we can build a matrix manually and we're also going to see how we can build a matrix that is prefilled with auto-incrementing values.

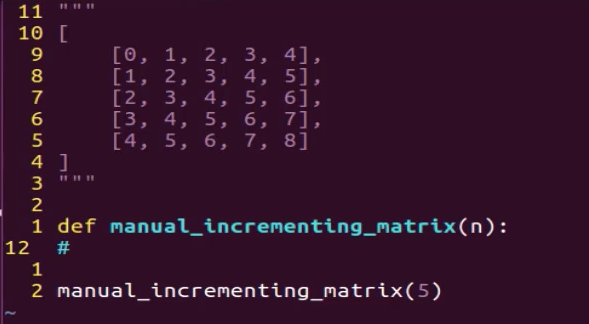
## Guide Tasks

If that sounds a little confusing right here in this comment I have an example of what we're looking for our function to be able to do.

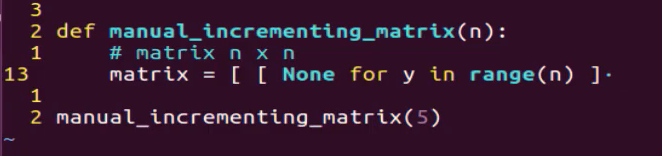


I have a list that contains lists inside of it and this is a traditional matrix so I have a list and inside of it there are five list items and then inside of each one of those lists are five elements so we have here 1 2 3 and 4 starting with the zeroth index and that by itself would not be the most challenging thing to do if we were to not care about how we are pre-filling this but where it gets tricky is being able to auto-fill it and increment it. So we're not simply trying to create values that go from 0 to 4. We want the next element to automatically increment. So the very first element in the second list is going to be one and the third one it's going to be two so on and so forth and we also can't hard code so the function that we're going to build out has to be able to perform this if we wanted a matrix that's 5 by 5 but if we wanted a matrix it was 10 by 10 our system would also have to do that as well. And so if you're following along and you're going through the exercise I want you to pause the video right now and then go and attempt to build out a function that will output these exact values and just so you have an idea of what that will look like. You should be able to call a function like this where you can name it whatever you want. I'm going to be descriptive with my name so I'll say manual\_incrementing\_matrix. And if I pass in the value 5 right here it should output this exact matrix and so it's going to include the first element that's going to start with the integer 0 and then count all the way up through whatever the last element is and then so on and so forth for each one of these items. So if you're following along and you're wanting to actually build this out and work on your skills then pause the video now and work on building this out.

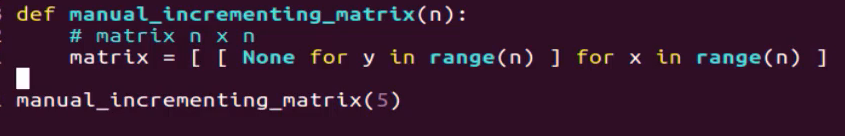
Now I'm going to walk through my own solution for this so I'm going to create a function here and I'm going to as you saw and name it manual\_incrementing\_matrix and I'm going to pass in an argument of n so this allows us to be very dynamic with the type of matrix that we're going to build. So if I pass in 5 it will give us what we have up here in the comment if I pass in 100 it will create this gigantic matrix so that is the way that I'm going to define it.



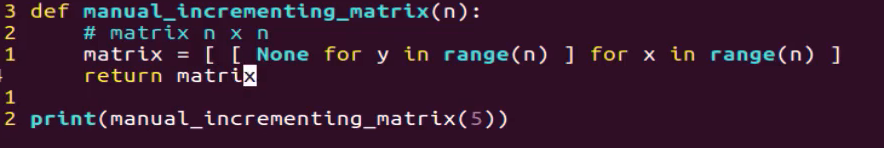
Now the very first step and let's actually outline in some comments what we want to do. The very first step is that I want to be able to build a matrix and it's going to start off with empty values so this should have a matrix of n x n in size and so that's going to allow me to build essentially what we have just right up here. Now there are a few ways that you could do this in Python but I really like the syntax of being able to use a list comprehension and so what I can do is create a variable here called matrix and then I'm going to use a nested list comprehension. If you've never used them before or if you've used them in limited types of situations where this is going to allow me to do is to pass in a few different processes inside of here and it's going to build the list for us and in this case, it's going to build a list that contains multiple lists inside of it. So here I'm going to say matrix equals and then go with two brackets and then I want to prefilled this first just with empty values so I'll say none for y in range and then I'm going to pass in the range which is going to be n so this is going to allow us to build a set of empty values and it's going to define exactly how many it's going to create for us. Let's close off that first list comprehension and then we're going to have a second one because as you can see from above we are going for a nested matrix.



I'm going to say now for x in range n because this is going to be the end and then I can close this off.



If that looks a little bit weird for you let's just take a quick break here and let's see exactly what this will give us. I'm going to come right here and let's just print it out and I'm going to save this file. Now if I run this and I don't have any kinds of spelling errors or anything like that we have none. And that's my fault because I'm actually not returning that ends. That's a little mistake. So right down here I need to say return matrix. Now if I save it and run it there we go. That's perfect.

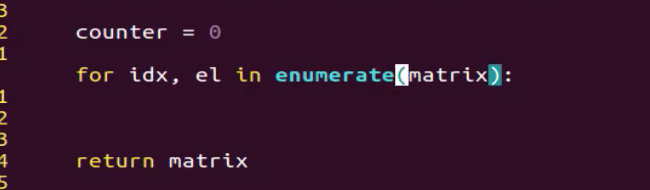


So as you can see right here what we have is our basic empty matrix it starts with a list that wraps the entire system up and then we have one element which contains five empty values. Then we have another one that contains five more and so on and so forth. And if you see we have 1 2 3 4 and 5. So we've already created our matrix and so that is one part.

large

Now we need to go and we actually need to populate it. So this is going to be a little bit more where the tricky side comes in and I know if you're watching and you've never used a list comprehension you may think that this was the tricky part but once you have worked with that and practiced it a while this is actually the easier part. Once we get down here this is where the tricky side of it is going to come in.

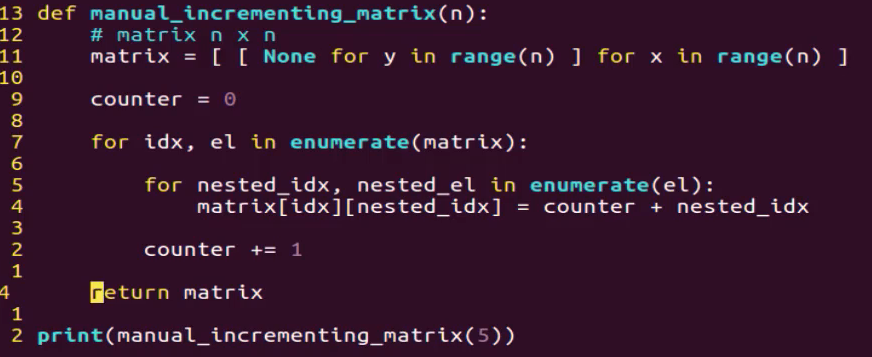
So the first thing I'm going to do because if you notice that type of behavior we're looking for we are looking to increment the count here. So we're starting with zero and then the little trend that I see here. When I was asked to build this out for students exercise what I noticed right away was that we have a pattern and not just a numerical incrementing pattern but also one thing you may notice is we have a pattern where the first element in each one of these rows and each one of these lists is also the index. So because this is the very first list in our parent list its first element is zero and then here the index of this first one is one and so on and so forth and if we had a hundred then the very last one would be ninety-nine. That would be the first way that it starts off. So that gives me a hint on exactly how this should be built out. And so what we're going to need to do is to first keep track of a number. So I'm going to create a counter variable here and I'm going to start off by setting it to zero. And let's give ourselves a little bit more room so we can see what's going on. And now I know I'm going to use a nested for loop so I can say four and I need to keep track of the index and so we can do that by saying for idx which is just short for index. For index L which the L is short for the element in and you may be able to think you may think that we can just say matrix and then start our loop. However, what I need to do is I need to keep track of this index value and the way that you can do that in Python is you need to first cast this matrix to the enumerate function. So I am gonna say in enumerate just like that.



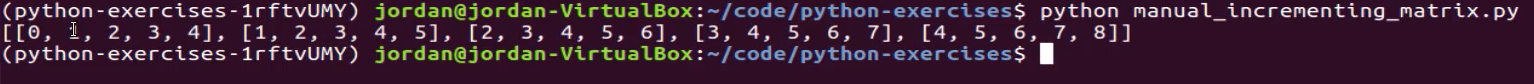
So what that's going to allow me to do is normally with a for loop it would say for L in Matrix and that would allow me to have access to this element then to the second one and so on. However, what I want is I don't want just the element. I also want to know what index I'm on because remember that is how I know what to start our system off with.

So that is very important and something I come down here and start a second loop. So there's going to be a nested loop and I'll say 'for' and this is also going to use the enumerate say for nested\_idx, nested el so that simply means that because we have this nested loop here this nested set of lists. The first time I am jumping into the matrix and I'm grabbing the first element. Now we need to iterate through the nested elements so in this case, nested\_idx is going to refer to this zero, and the first time it goes through and the nested element is going to be the actual value. And so that is keeping track of our index and the value. And so I say nested\_el in enumerate(el) and this is going to be in the not actually the matrix. I almost made a mistake there. We are wanting to loop over the element itself. So now instead of just going through and saying okay I'm going to throw the first one the second one. Now we're going to go through 0 1 2 3 and go through each one of those. So now that we have that we need to do is to assign the value. So I'm going to take the matrix and then grab the index and this is the tricky part. And if you tried to go through this yourself and you couldn't get it working what I've seen is this is the part that is the most confusing and that is where you need to actually go and you need to dive into multiple sets of these collections so it's not enough to just say matrix and then grab the index because what that's going to do is it's going to grab the first item but it's going to grab the entire first item. We need to also grab the index of the nested items I am gonna say nested\_idx.

Now we can assign the calendar which starts at zero and then we're going to add nested\_idx on top of that. From there we need to make sure we're going to have to break out of this for a loop. And so then I can say counter increment by 1 and then at the very end, it is going to just return that matrix value.



So let's save this and see if it's working and I believe I have everything there that's needed. So as a python manual incrementing matrix. And that is working perfectly as you can see we have 5 elements 0 1 2 3 and 4 and the first one 1 2 3 4 and the second one and it goes all the way through.

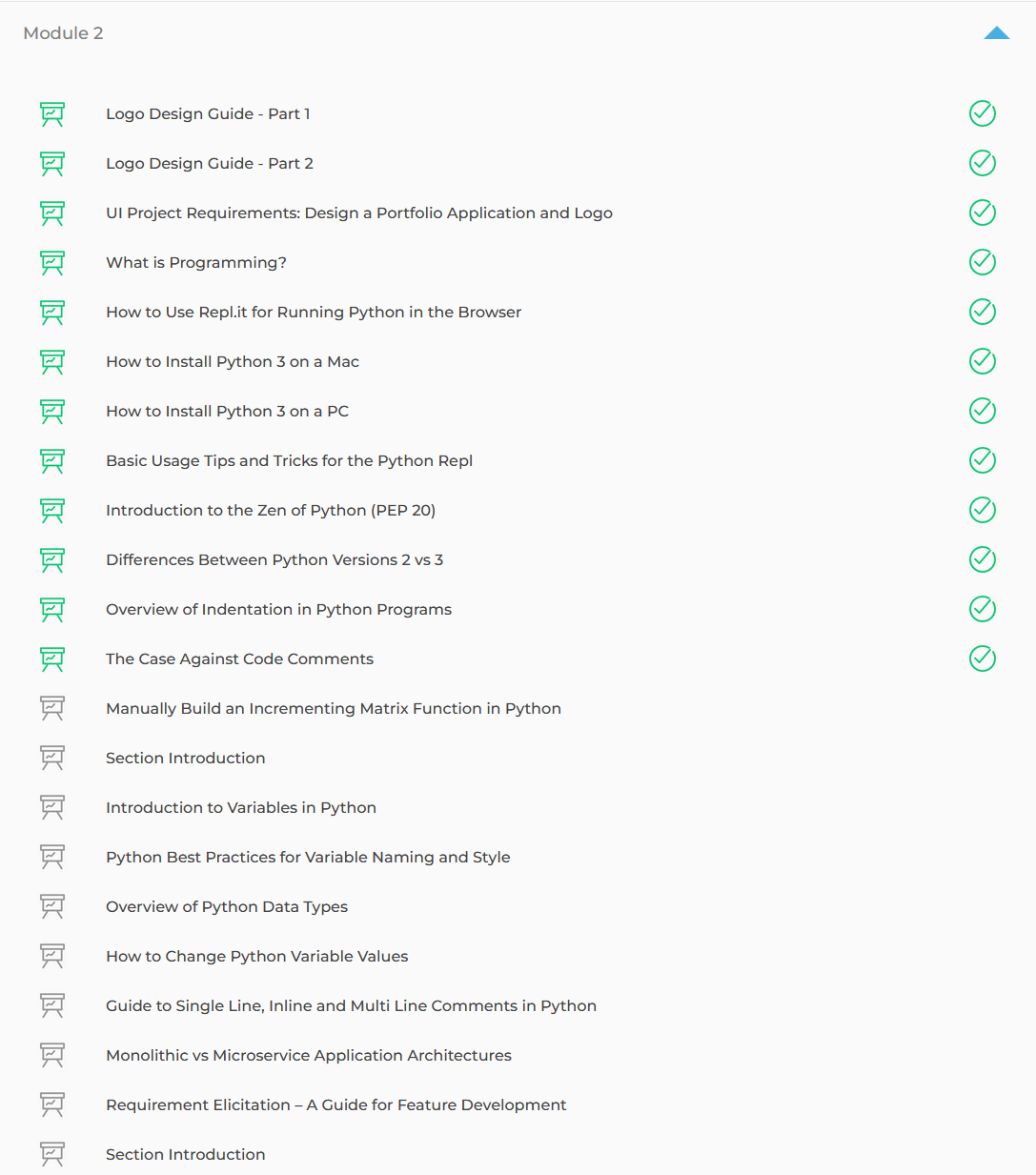


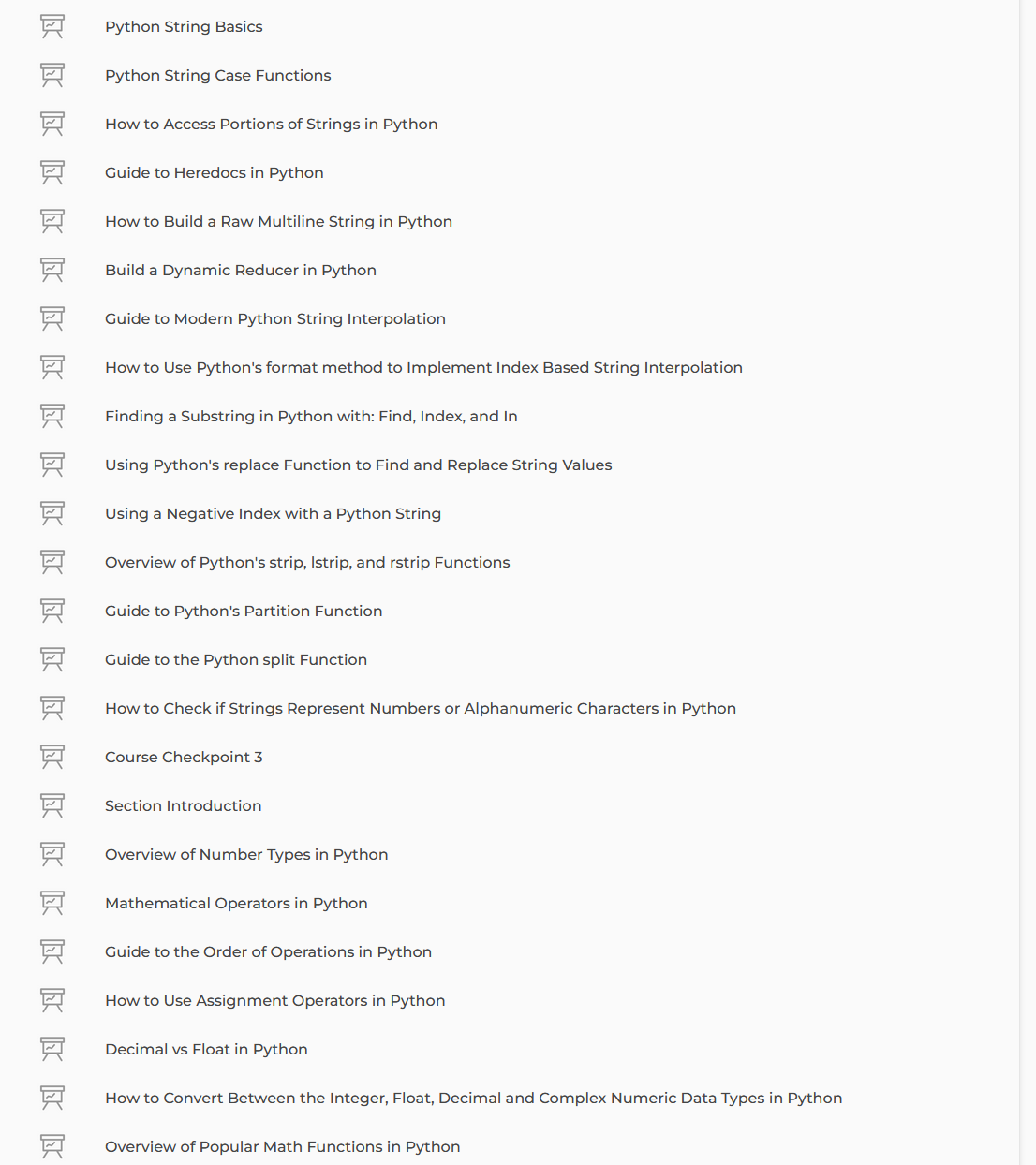
This perfectly matches up with what we had up here in the comments so we've effectively built out a full functioning matrix and we did it all without numpy or any tools like that. If I change this to ten come and run it again you can see that it does the exact same thing. Gives us a matrix that contains 10 elements and then each one of those items inside has 10 elements as well and it all auto increments. Now let's just kind of take one last review in case any of that was confusing.

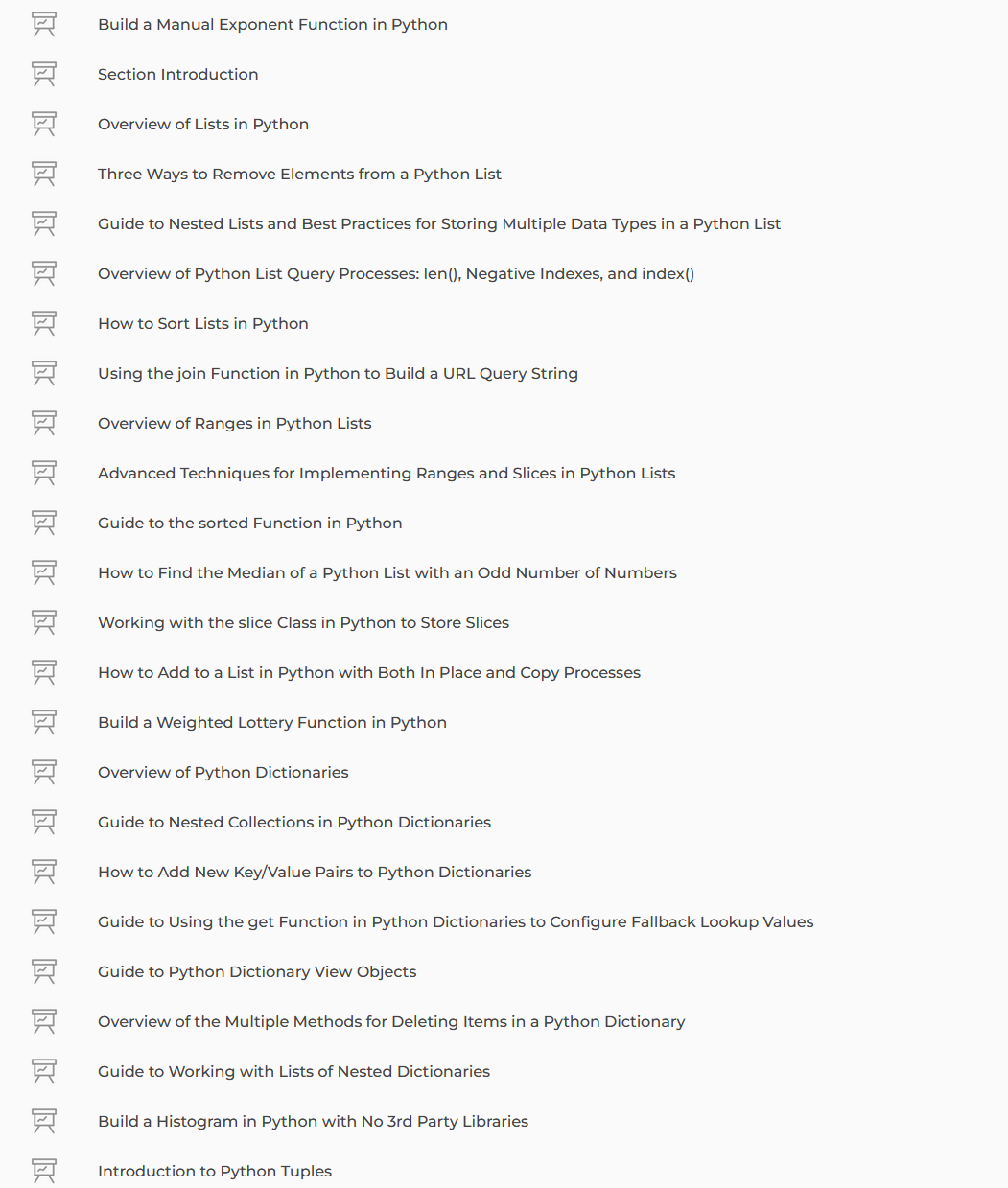
So the very first thing we did was we created a function and the function takes in how many items that you want the matrix to be. So it will generate a set of items. Then each one of those nested lists will contain the same number of items and that's doing exactly what we're asked to do right up here. The first step we took was to generate an empty matrix and the reason why I went with that approach was because it was a lot easier to have a working data structure and to be able to simply iterate over it and then slide the values in that we needed. And so here we created this empty matrix and we used a list comprehension where we were able to take this first nested item and we said okay I want you to add the value of none for and Y was just our block variable. And so he said for y in range n which just means I want you to loop over this range. And so when we pass 5 n that just means I want you to loop over from 1 to 5. I want you to go through this five times and then for each time I want you to assign the value of none and then I want you to go and do that again for the parent structure and that's how we were able to create five elements just like this or n elements and then go inside and create 5 elements or n elements inside each one of those items from there we created a counter set it equal to zero because this was kind of the trick we were able to use because the counter is the way we are able to increment our values.

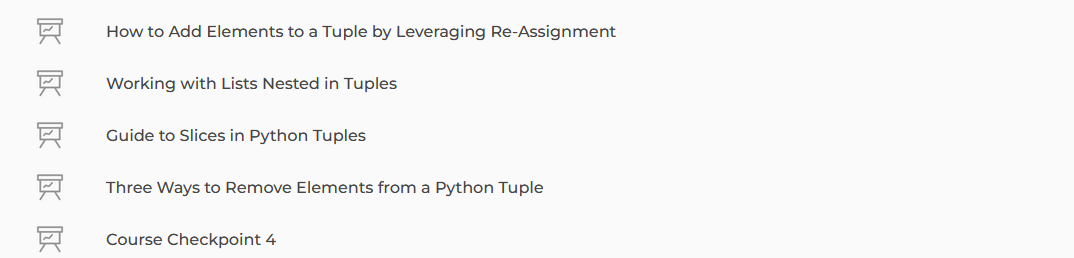
So from there we use the enumerate function or we cast the matrix so that we could use this cool little trick where we can grab the index and we could loop over and have access to the element. From there we created a nested loop inside of that where we also used enumerate. We said okay I want to have the index so in this case that that nested IDX in say third or fourth item here this would have been 0 here and then it would be one here and then 2 and it would keep on going like that. So we had access to know what index we were on and then we also knew the item. So this would not start off at zero for this item, in fact, this one would just give us access and say okay that in that case that one was 3 then that one is 4 then that is 5. That's how we're able to go in and then assign those so we're looping over that and then this is where the assignment happens so we say okay we're grabbing the matrix the IDX value. This is what we're getting IDX from. So I'm looping over grabbing that index and then we're using the double bracket syntax so we have access to the nested index as well. And then all we're doing is we're taking our counter and then we're adding to whatever the nested index is and that's how we were able to implement that trick where we were able to increment it with each new item that we were going through. Then in order to make sure that each new one started with one higher element or an element that had a value greater than one then we were able to do that right here. So at the very end, we simply return the matrix and that is it.

So I hope that you enjoyed that. That was not an easy function to build out. So if there are some elements here there are confusing to you or you were not sure exactly how it worked or you weren't able to get it working the first time. Don't worry. That was definitely a more challenging system to build out. So I recommend going through it a few times becoming familiar with each one of the components that was implemented and I think that you'll find that these are some skills that are really going to help you as you're building out your own Python programs.









PEP8

NAMING CONVENTIONS

## [Naming Conventions](https://peps.python.org/pep-0008/#naming-conventions)

The naming conventions of Python’s library are a bit of a mess, so we’ll never get this completely consistent – nevertheless, here are the currently recommended naming standards. New modules and packages (including third party frameworks) should be written to these standards, but where an existing library has a different style, internal consistency is preferred.

### [Overriding Principle](https://peps.python.org/pep-0008/#overriding-principle)

Names that are visible to the user as public parts of the API should follow conventions that reflect usage rather than implementation.

### [Descriptive: Naming Styles](https://peps.python.org/pep-0008/#descriptive-naming-styles)

There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for.

The following naming styles are commonly distinguished:

* b (single lowercase letter)
* B (single uppercase letter)
* lowercase
* lower\_case\_with\_underscores
* UPPERCASE
* UPPER\_CASE\_WITH\_UNDERSCORES
* CapitalizedWords (or CapWords, or CamelCase – so named because of the bumpy look of its letters [[4]](https://peps.python.org/pep-0008/#id8)). This is also sometimes known as StudlyCaps.

Note: When using acronyms in CapWords, capitalize all the letters of the acronym. Thus HTTPServerError is better than HttpServerError.

* mixedCase (differs from CapitalizedWords by initial lowercase character!)
* Capitalized\_Words\_With\_Underscores (ugly!)

There’s also the style of using a short unique prefix to group related names together. This is not used much in Python, but it is mentioned for completeness. For example, the os.stat() function returns a tuple whose items traditionally have names like st\_mode, st\_size, st\_mtime and so on. (This is done to emphasize the correspondence with the fields of the POSIX system call struct, which helps programmers familiar with that.)

The X11 library uses a leading X for all its public functions. In Python, this style is generally deemed unnecessary because attribute and method names are prefixed with an object, and function names are prefixed with a module name.

In addition, the following special forms using leading or trailing underscores are recognized (these can generally be combined with any case convention):

* \_single\_leading\_underscore: weak “internal use” indicator. E.g. from M import \* does not import objects whose names start with an underscore.
* single\_trailing\_underscore\_: used by convention to avoid conflicts with Python keyword, e.g. :
* tkinter.Toplevel(master, class\_='ClassName')
* \_\_double\_leading\_underscore: when naming a class attribute, invokes name mangling (inside class FooBar, \_\_boo becomes \_FooBar\_\_boo; see below).
* \_\_double\_leading\_and\_trailing\_underscore\_\_: “magic” objects or attributes that live in user-controlled namespaces. E.g. \_\_init\_\_, \_\_import\_\_ or \_\_file\_\_. Never invent such names; only use them as documented.

### [Prescriptive: Naming Conventions](https://peps.python.org/pep-0008/#prescriptive-naming-conventions)

#### [Names to Avoid](https://peps.python.org/pep-0008/#names-to-avoid)

Never use the characters ‘l’ (lowercase letter el), ‘O’ (uppercase letter oh), or ‘I’ (uppercase letter eye) as single character variable names.

In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use ‘l’, use ‘L’ instead.

#### [ASCII Compatibility](https://peps.python.org/pep-0008/#ascii-compatibility)

Identifiers used in the standard library must be ASCII compatible as described in the [policy section](https://peps.python.org/pep-3131/#policy-specification) of [PEP 3131](https://peps.python.org/pep-3131/).

#### [Package and Module Names](https://peps.python.org/pep-0008/#package-and-module-names)

Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

When an extension module written in C or C++ has an accompanying Python module that provides a higher level (e.g. more object oriented) interface, the C/C++ module has a leading underscore (e.g. \_socket).

#### [Class Names](https://peps.python.org/pep-0008/#class-names)

Class names should normally use the CapWords convention.

The naming convention for functions may be used instead in cases where the interface is documented and used primarily as a callable.

Note that there is a separate convention for builtin names: most builtin names are single words (or two words run together), with the CapWords convention used only for exception names and builtin constants.

#### [Type Variable Names](https://peps.python.org/pep-0008/#type-variable-names)

Names of type variables introduced in [PEP 484](https://peps.python.org/pep-0484/) should normally use CapWords preferring short names: T, AnyStr, Num. It is recommended to add suffixes \_co or \_contra to the variables used to declare covariant or contravariant behavior correspondingly:

from typing import TypeVar

VT\_co = TypeVar('VT\_co', covariant=True)

KT\_contra = TypeVar('KT\_contra', contravariant=True)

#### [Exception Names](https://peps.python.org/pep-0008/#exception-names)

Because exceptions should be classes, the class naming convention applies here. However, you should use the suffix “Error” on your exception names (if the exception actually is an error).

#### [Global Variable Names](https://peps.python.org/pep-0008/#global-variable-names)

(Let’s hope that these variables are meant for use inside one module only.) The conventions are about the same as those for functions.

Modules that are designed for use via from M import \* should use the \_\_all\_\_ mechanism to prevent exporting globals, or use the older convention of prefixing such globals with an underscore (which you might want to do to indicate these globals are “module non-public”).

#### [Function and Variable Names](https://peps.python.org/pep-0008/#function-and-variable-names)

Function names should be lowercase, with words separated by underscores as necessary to improve readability.

Variable names follow the same convention as function names.

mixedCase is allowed only in contexts where that’s already the prevailing style (e.g. threading.py), to retain backwards compatibility.

#### [Function and Method Arguments](https://peps.python.org/pep-0008/#function-and-method-arguments)

Always use self for the first argument to instance methods.

Always use cls for the first argument to class methods.

If a function argument’s name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus class\_ is better than clss. (Perhaps better is to avoid such clashes by using a synonym.)

#### [Method Names and Instance Variables](https://peps.python.org/pep-0008/#method-names-and-instance-variables)

Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability.

Use one leading underscore only for non-public methods and instance variables.

To avoid name clashes with subclasses, use two leading underscores to invoke Python’s name mangling rules.

Python mangles these names with the class name: if class Foo has an attribute named \_\_a, it cannot be accessed by Foo.\_\_a. (An insistent user could still gain access by calling Foo.\_Foo\_\_a.) Generally, double leading underscores should be used only to avoid name conflicts with attributes in classes designed to be subclassed.

Note: there is some controversy about the use of \_\_names (see below).

#### [Constants](https://peps.python.org/pep-0008/#constants)

Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include MAX\_OVERFLOW and TOTAL.

#### [Designing for Inheritance](https://peps.python.org/pep-0008/#designing-for-inheritance)

Always decide whether a class’s methods and instance variables (collectively: “attributes”) should be public or non-public. If in doubt, choose non-public; it’s easier to make it public later than to make a public attribute non-public.

Public attributes are those that you expect unrelated clients of your class to use, with your commitment to avoid backwards incompatible changes. Non-public attributes are those that are not intended to be used by third parties; you make no guarantees that non-public attributes won’t change or even be removed.

We don’t use the term “private” here, since no attribute is really private in Python (without a generally unnecessary amount of work).

Another category of attributes are those that are part of the “subclass API” (often called “protected” in other languages). Some classes are designed to be inherited from, either to extend or modify aspects of the class’s behavior. When designing such a class, take care to make explicit decisions about which attributes are public, which are part of the subclass API, and which are truly only to be used by your base class.

With this in mind, here are the Pythonic guidelines:

* Public attributes should have no leading underscores.
* If your public attribute name collides with a reserved keyword, append a single trailing underscore to your attribute name. This is preferable to an abbreviation or corrupted spelling. (However, notwithstanding this rule, ‘cls’ is the preferred spelling for any variable or argument which is known to be a class, especially the first argument to a class method.)

Note 1: See the argument name recommendation above for class methods.

* For simple public data attributes, it is best to expose just the attribute name, without complicated accessor/mutator methods. Keep in mind that Python provides an easy path to future enhancement, should you find that a simple data attribute needs to grow functional behavior. In that case, use properties to hide functional implementation behind simple data attribute access syntax.

Note 1: Try to keep the functional behavior side-effect free, although side-effects such as caching are generally fine.

Note 2: Avoid using properties for computationally expensive operations; the attribute notation makes the caller believe that access is (relatively) cheap.

* If your class is intended to be subclassed, and you have attributes that you do not want subclasses to use, consider naming them with double leading underscores and no trailing underscores. This invokes Python’s name mangling algorithm, where the name of the class is mangled into the attribute name. This helps avoid attribute name collisions should subclasses inadvertently contain attributes with the same name.

Note 1: Note that only the simple class name is used in the mangled name, so if a subclass chooses both the same class name and attribute name, you can still get name collisions.

Note 2: Name mangling can make certain uses, such as debugging and \_\_getattr\_\_(), less convenient. However the name mangling algorithm is well documented and easy to perform manually.

Note 3: Not everyone likes name mangling. Try to balance the need to avoid accidental name clashes with potential use by advanced callers.

### [Public and Internal Interfaces](https://peps.python.org/pep-0008/#public-and-internal-interfaces)

Any backwards compatibility guarantees apply only to public interfaces. Accordingly, it is important that users be able to clearly distinguish between public and internal interfaces.

Documented interfaces are considered public, unless the documentation explicitly declares them to be provisional or internal interfaces exempt from the usual backwards compatibility guarantees. All undocumented interfaces should be assumed to be internal.

To better support introspection, modules should explicitly declare the names in their public API using the \_\_all\_\_ attribute. Setting \_\_all\_\_ to an empty list indicates that the module has no public API.

Even with \_\_all\_\_ set appropriately, internal interfaces (packages, modules, classes, functions, attributes or other names) should still be prefixed with a single leading underscore.

An interface is also considered internal if any containing namespace (package, module or class) is considered internal.

Imported names should always be considered an implementation detail. Other modules must not rely on indirect access to such imported names unless they are an explicitly documented part of the containing module’s API, such as os.path or a package’s \_\_init\_\_ module that exposes functionality from submodules.

## [Programming Recommendations](https://peps.python.org/pep-0008/#programming-recommendations)

* Code should be written in a way that does not disadvantage other implementations of Python (PyPy, Jython, IronPython, Cython, Psyco, and such).

For example, do not rely on CPython’s efficient implementation of in-place string concatenation for statements in the form a += b or a = a + b. This optimization is fragile even in CPython (it only works for some types) and isn’t present at all in implementations that don’t use refcounting. In performance sensitive parts of the library, the ''.join() form should be used instead. This will ensure that concatenation occurs in linear time across various implementations.

* Comparisons to singletons like None should always be done with is or is not, never the equality operators.

Also, beware of writing if x when you really mean if x is not None – e.g. when testing whether a variable or argument that defaults to None was set to some other value. The other value might have a type (such as a container) that could be false in a boolean context!

* Use is not operator rather than not ... is. While both expressions are functionally identical, the former is more readable and preferred:
* # Correct:
* if foo is not None:
* # Wrong:
* if not foo is None:
* When implementing ordering operations with rich comparisons, it is best to implement all six operations (\_\_eq\_\_, \_\_ne\_\_, \_\_lt\_\_, \_\_le\_\_, \_\_gt\_\_, \_\_ge\_\_) rather than relying on other code to only exercise a particular comparison.

To minimize the effort involved, the functools.total\_ordering() decorator provides a tool to generate missing comparison methods.

[PEP 207](https://peps.python.org/pep-0207/) indicates that reflexivity rules are assumed by Python. Thus, the interpreter may swap y > x with x < y, y >= x with x <= y, and may swap the arguments of x == y and x != y. The sort() and min() operations are guaranteed to use the < operator and the max() function uses the > operator. However, it is best to implement all six operations so that confusion doesn’t arise in other contexts.

* Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier:
* # Correct:
* def f(x): return 2\*x
* # Wrong:
* f = lambda x: 2\*x

The first form means that the name of the resulting function object is specifically ‘f’ instead of the generic ‘<lambda>’. This is more useful for tracebacks and string representations in general. The use of the assignment statement eliminates the sole benefit a lambda expression can offer over an explicit def statement (i.e. that it can be embedded inside a larger expression)

* Derive exceptions from Exception rather than BaseException. Direct inheritance from BaseException is reserved for exceptions where catching them is almost always the wrong thing to do.

Design exception hierarchies based on the distinctions that code catching the exceptions is likely to need, rather than the locations where the exceptions are raised. Aim to answer the question “What went wrong?” programmatically, rather than only stating that “A problem occurred” (see [PEP 3151](https://peps.python.org/pep-3151/) for an example of this lesson being learned for the builtin exception hierarchy)

Class naming conventions apply here, although you should add the suffix “Error” to your exception classes if the exception is an error. Non-error exceptions that are used for non-local flow control or other forms of signaling need no special suffix.

* Use exception chaining appropriately. raise X from Y should be used to indicate explicit replacement without losing the original traceback.

When deliberately replacing an inner exception (using raise X from None), ensure that relevant details are transferred to the new exception (such as preserving the attribute name when converting KeyError to AttributeError, or embedding the text of the original exception in the new exception message).

* When catching exceptions, mention specific exceptions whenever possible instead of using a bare except: clause:
* try:
* import platform\_specific\_module
* except ImportError:
* platform\_specific\_module = None

A bare except: clause will catch SystemExit and KeyboardInterrupt exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use except Exception: (bare except is equivalent to except BaseException:).

A good rule of thumb is to limit use of bare ‘except’ clauses to two cases:

* 1. If the exception handler will be printing out or logging the traceback; at least the user will be aware that an error has occurred.
  2. If the code needs to do some cleanup work, but then lets the exception propagate upwards with raise. try...finally can be a better way to handle this case.
* When catching operating system errors, prefer the explicit exception hierarchy introduced in Python 3.3 over introspection of errno values.
* Additionally, for all try/except clauses, limit the try clause to the absolute minimum amount of code necessary. Again, this avoids masking bugs:
* # Correct:
* try:
* value = collection[key]
* except KeyError:
* return key\_not\_found(key)
* else:
* return handle\_value(value)
* # Wrong:
* try:
* # Too broad!
* return handle\_value(collection[key])
* except KeyError:
* # Will also catch KeyError raised by handle\_value()
* return key\_not\_found(key)
* When a resource is local to a particular section of code, use a with statement to ensure it is cleaned up promptly and reliably after use. A try/finally statement is also acceptable.
* Context managers should be invoked through separate functions or methods whenever they do something other than acquire and release resources:
* # Correct:
* with conn.begin\_transaction():
* do\_stuff\_in\_transaction(conn)
* # Wrong:
* with conn:
* do\_stuff\_in\_transaction(conn)

The latter example doesn’t provide any information to indicate that the \_\_enter\_\_ and \_\_exit\_\_ methods are doing something other than closing the connection after a transaction. Being explicit is important in this case.

* Be consistent in return statements. Either all return statements in a function should return an expression, or none of them should. If any return statement returns an expression, any return statements where no value is returned should explicitly state this as return None, and an explicit return statement should be present at the end of the function (if reachable):
* # Correct:
* def foo(x):
* if x >= 0:
* return math.sqrt(x)
* else:
* return None
* def bar(x):
* if x < 0:
* return None
* return math.sqrt(x)
* # Wrong:
* def foo(x):
* if x >= 0:
* return math.sqrt(x)
* def bar(x):
* if x < 0:
* return
* return math.sqrt(x)
* Use ''.startswith() and ''.endswith() instead of string slicing to check for prefixes or suffixes.

startswith() and endswith() are cleaner and less error prone:

# Correct:

if foo.startswith('bar'):

# Wrong:

if foo[:3] == 'bar':

* Object type comparisons should always use isinstance() instead of comparing types directly:
* # Correct:
* if isinstance(obj, int):
* # Wrong:
* if type(obj) is type(1):
* For sequences, (strings, lists, tuples), use the fact that empty sequences are false:
* # Correct:
* if not seq:
* if seq:
* # Wrong:
* if len(seq):
* if not len(seq):
* Don’t write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors (or more recently, reindent.py) will trim them.
* Don’t compare boolean values to True or False using ==:
* # Correct:
* if greeting:
* # Wrong:
* if greeting == True:

Worse:

# Wrong:

if greeting is True:

* Use of the flow control statements return/break/continue within the finally suite of a try...finally, where the flow control statement would jump outside the finally suite, is discouraged. This is because such statements will implicitly cancel any active exception that is propagating through the finally suite:
* # Wrong:
* def foo():
* try:
* 1 / 0
* finally:
* return 42

### [Function Annotations](https://peps.python.org/pep-0008/#function-annotations)

With the acceptance of [PEP 484](https://peps.python.org/pep-0484/), the style rules for function annotations have changed.

* Function annotations should use [PEP 484](https://peps.python.org/pep-0484/) syntax (there are some formatting recommendations for annotations in the previous section).
* The experimentation with annotation styles that was recommended previously in this PEP is no longer encouraged.
* However, outside the stdlib, experiments within the rules of [PEP 484](https://peps.python.org/pep-0484/) are now encouraged. For example, marking up a large third party library or application with [PEP 484](https://peps.python.org/pep-0484/) style type annotations, reviewing how easy it was to add those annotations, and observing whether their presence increases code understandability.
* The Python standard library should be conservative in adopting such annotations, but their use is allowed for new code and for big refactorings.
* For code that wants to make a different use of function annotations it is recommended to put a comment of the form:
* # type: ignore

near the top of the file; this tells type checkers to ignore all annotations. (More fine-grained ways of disabling complaints from type checkers can be found in [PEP 484](https://peps.python.org/pep-0484/).)

* Like linters, type checkers are optional, separate tools. Python interpreters by default should not issue any messages due to type checking and should not alter their behavior based on annotations.
* Users who don’t want to use type checkers are free to ignore them. However, it is expected that users of third party library packages may want to run type checkers over those packages. For this purpose [PEP 484](https://peps.python.org/pep-0484/) recommends the use of stub files: .pyi files that are read by the type checker in preference of the corresponding .py files. Stub files can be distributed with a library, or separately (with the library author’s permission) through the typeshed repo [[5]](https://peps.python.org/pep-0008/#id9).

### [Variable Annotations](https://peps.python.org/pep-0008/#variable-annotations)

[PEP 526](https://peps.python.org/pep-0526/) introduced variable annotations. The style recommendations for them are similar to those on function annotations described above:

* Annotations for module level variables, class and instance variables, and local variables should have a single space after the colon.
* There should be no space before the colon.
* If an assignment has a right hand side, then the equality sign should have exactly one space on both sides:
* # Correct:
* code: int
* class Point:
* coords: Tuple[int, int]
* label: str = '<unknown>'
* # Wrong:
* code:int # No space after colon
* code : int # Space before colon
* class Test:
* result: int=0 # No spaces around equality sign
* Although the [PEP 526](https://peps.python.org/pep-0526/) is accepted for Python 3.6, the variable annotation syntax is the preferred syntax for stub files on all versions of Python (see [PEP 484](https://peps.python.org/pep-0484/) for details).