Assignment 1: Dynamo of Volition

This assignment is due on September 23rd at 11:59pm.

Dynamo of Volition

Static scoping, also known as lexical scoping, is by far the most commonly used technique for resolving the binding of names to variables in a programming language. All major languages (with the exception of **Perl** (https://perldoc.perl.org/perlfaq7#What's-the-difference-between-dynamic-and-lexical-(static)-scoping?-Between-local()-and-my()?) and **Bash** (https://www.gnu.org/savannah-checkouts/gnu/bash/manual/bash.html#Shell-Functions)) use static scoping. Lexical scoping is so common that its algorithm for resolving the binding of names to variables feels intuitive to most programmers. It almost feels like we want to say, "What other possible way is there?"

And yet, there is another way! Dynamic scoping! Even though static scoping feels so right, because we are going to implement dynamic scoping in this assignment, it helps to be explicit about how the binding of names to variables works under static scoping. First, let's define some terms. *Note*: All of these definitions are from the perspective of a particular statement s in program code.

0. local scope of s: the scope of s.
1. static parent scope of s: the local scope of the statement that created the local scope of s.
2. static ancestor scope of s: one of the static parent scopes of s.
3. static ancestor scopes of s: all of the static ancestor scopes of s.

Assume that we want to resolve the binding of the name x to a variable. The algorithm for performing that binding at a statement s in a program has 4 steps:

1. If x names a local variable, x is bound to that variable.
2. If x does not name a local variable, consider the static parent scope of s, p.
3. If the scope p contains a local variable named x, x is bound to that variable.
4. If the name x is still not bound to a variable, consider the static parent scope of p. Continue searching static parent scopes until there are no more static ancestors.

In this assignment we will assume that there is an ultimate *global* scope that encompasses all other scopes. To make the algorithm more concrete, consider the following Python code:

```
def outer():
  a = "outer contents of a"
  b = "outer contents of b"
  c = "outer contents of c"
```

```
  def inner():
    a = "inner contents of a"
    b = "inner contents of b"
    def inner_inner():
      a = "inner inner contents of a"
      print(f"{a=}")
      print(f"{b=}")
      print(f"{c=}")
    inner_inner()
  inner()
if __name__=="main":
  outer()
```

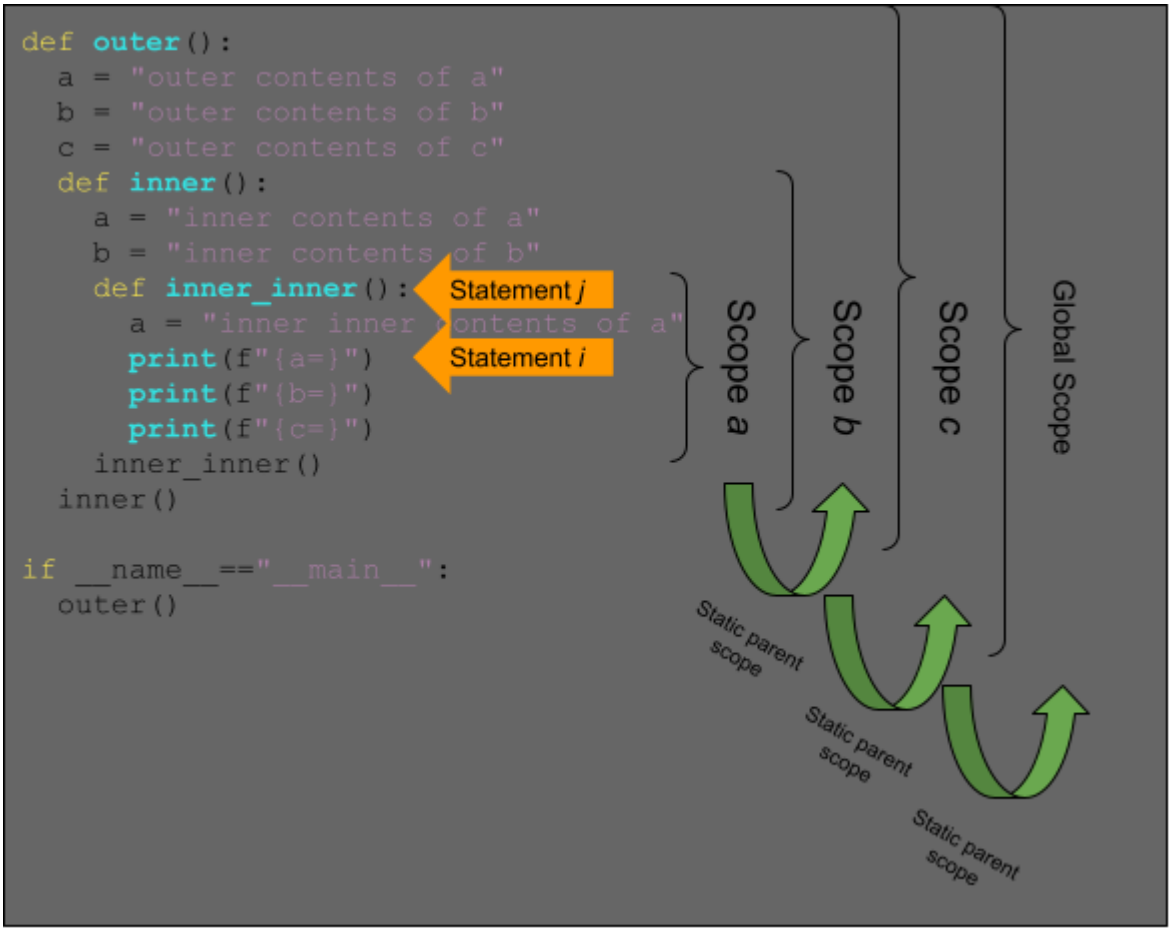
When Before continuing, we should be clear about the conditions under which a new *scope* is created in Python. The cool thing about searching for clarity on this issue is that we don't have to look very far! There is an entire subsection in the **Python Language Reference dedicated to just this topic (https://docs.python.org/3/reference/executionmodel.html#naming-and-binding)**. Here is the relevant snippet:

> Each assignment or import statement occurs within a block defined by a class or function definition or at the module level (the top-level code block).

To be clear, they are using the term *block* here the way that *Sebesta* uses the term (i.e., a section of code with its own, minimized scope [p. 213]). Therefore, the language documentation is telling us that new scopes are created every time there is a function or class (or module) defined.

(As a side note, I *highly recommend* reading that entire subsection. If you do, you will realize that what we are learning in this class has direct applicability for those who actually design and implement the programming languages that we use every day!)

Now we can get down to business!



When *Statement i* is encountered statically (i.e., when the program is parsed), the algorithm for resolving the binding between name a and a variable proceeds as follows:

1. The name a is bound to a local variable. The resolution is complete.

Pretty easy, right?

Now, what happens when the statement after *Statement i* is encountered statically and the binding between b and a variable needs to be resolved?

1. b is not bound to a local variable.
2. Resolution proceeds to search the static parent of *Scope a* for a local variable named b.
3. *Scope b* does contain a local variable named b. The name is bound to that local variable.

Just what we expected!

And, finally, what happens when the binding between the name c and a variable needs to be resolved?

1. c is not bound to a local variable.
2. Resolution proceeds to search the static parent of *Scope a* for a local variable named c.
3. *Scope b* does not contain a local variable named c.
4. Resolution proceeds to search the static parent of *Scope b* for a local variable named c.
5. *Scope c* does contain a local variable named c. The name is bound to that local variable.

Lots of work, but largely as expected! Given that, the program prints what we would expect:

```
a='inner inner contents of a'
b='inner contents of b'
c='outer contents of c'
```

Again, we programmers feel that sort of algorithm in our gut. The algorithm for resolving the binding between names and variables using *dynamic scoping* may not be as intuitive because we don't rely on it as often. So, let's be specific about the algorithm and the vocabulary:

0. local scope of s: the scope of s.
1. dynamic parent scope of s: the local scope of the statement that executed the block of code that contains s.
2. dynamic ancestor scope of s: one of the dynamic parent scopes of s.
3. dynamic ancestor scopes of s: all the dynamic parent scopes of s.

Assume that we want to resolve the binding of the name x to a variable using dynamic scoping. The algorithm for performing that binding at statement s in a program has 4 steps:

1. If x names a local variable, x is bound to that variable.
2. If x does not name a local variable, consider the dynamic parent scope of p, x.
3. If the scope x contains a local variable named x, x is bound to that variable.
4. If the name x is still not bound to a variable, consider the dynamic parent scope of x. Continue searching dynamic parent scopes until there are no more dynamic ancestors.

Remember how we defined *dynamic parent scope* in class? The dynamic parent scope, call it P, of a scope S is the local scope of the code that *started execution* of code in scope S. Pretty cool!

Let's make that visual using the following example:

```
def outer():
  a = "outer_a"
  b = "outer_b"
  c = "outer_c"

  def inner1():
    a = "inner1_a"
    b = "inner1_b"
    inner3()

  def inner2():
    a = "inner2_a"
    b = "inner2_b"
    inner3()

  def inner3():
    a = "inner3_a"
    print(f"{a=}")
    print(f"{b=}")
    print(f"{c=}")

  print("Calling inner1:")
  inner1()
```

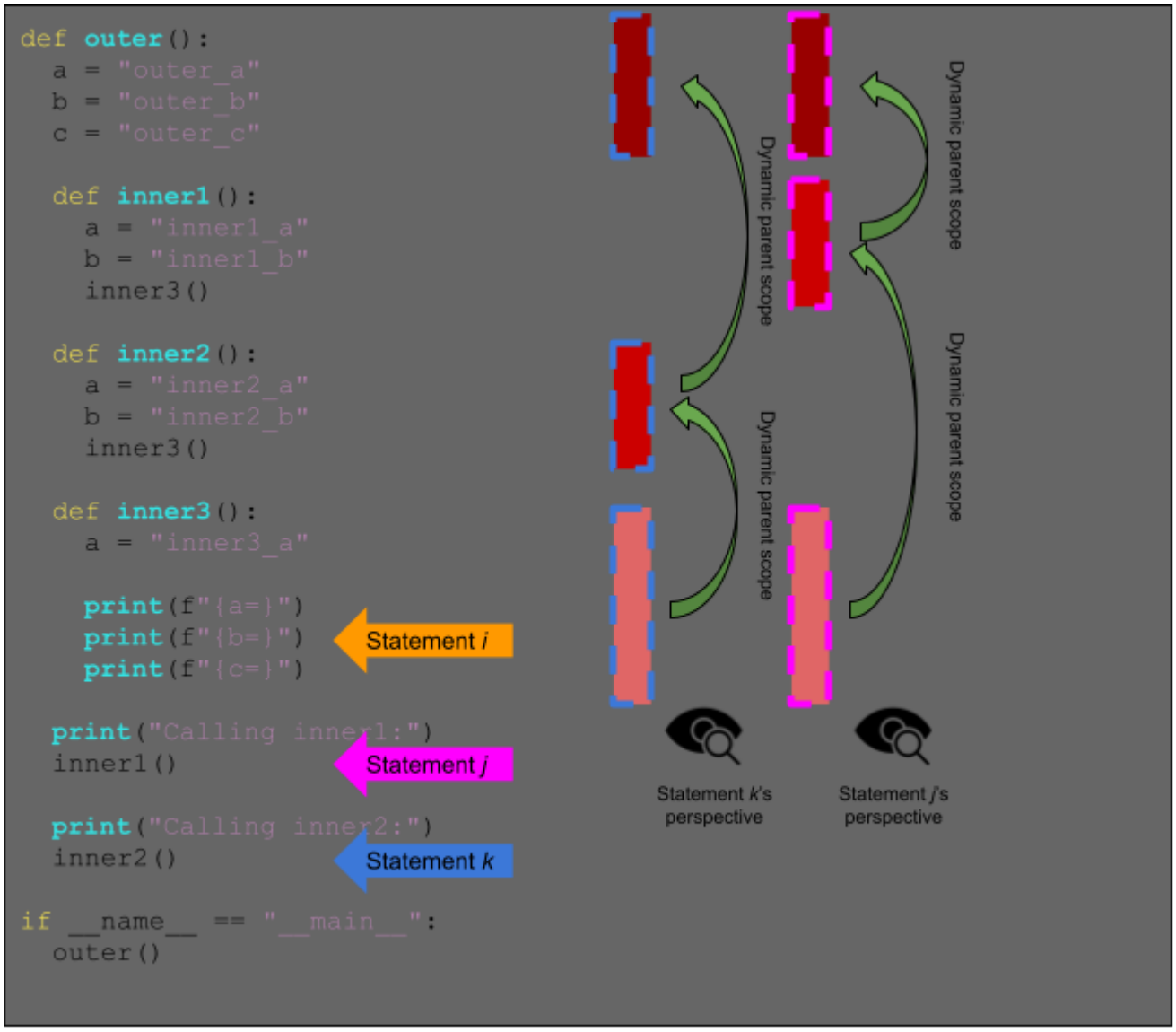
```
  print("Calling inner2:")
  inner2()

if __name__ == "__main__":
  outer()
```

When we let the Python interpreter do its usual work on this program, the output is

```
Calling inner1:
a='inner3_a'
b='outer_b'
c='outer_c'
Calling inner2:
a='inner3_a'
b='outer_b'
c='outer_c'
```

Not surprising!

What would be the output, however, if Python were dynamically scoped?



Eye by Vectorstall from NounProject.com

```
Calling inner1:
a='inner3_a'
b='inner1_b'
c='outer_c'
Calling inner2:
a='inner3_a'
b='inner2_b'
c='outer_c'
```

How does this work? What happens when *Statement i* is encountered at runtime and the binding between b and a variable needs to be resolved?

1. b is not a local variable.
2. Resolution proceeds to search the dynamic parent of *Statement i* for a local variable named b. Depending on whether `inner3` was called from `inner2` or `inner1`, the next scope to be searched will be different.
3. If `inner3` was called from `inner1` (*Statement j*), then the scope of that function is searched for a local variable named b. Success! b is bound to a local variable whose contents are `"inner1_b"`.
4. If `inner3` was called from `inner2` (*Statement k*), then the scope of that function is searched for a local variable named b. Success! b is bound to a local variable whose contents are `"inner2_b"`.

It sounds crazy, but it just works!

Making Python Dynamic-er

How can we make Python use dynamic scoping? Well, we aren't going to do it perfectly, but we'll get close.

Your task in this assignment is to implement a function named `get_dynamic_re` in a Python module named `dynamic_scope`. The function will return a `DynamicScope` object -- you will define and implement that class, too!

The `DynamicScope` class

The `DynamicScope` class will be used to encapsulate a *reference environment*, the names bound to variables (and their values) at a given statement, generated through dynamic scoping. The `DynamicScope` class will operate exactly like a Python `Mapping` -- the `Mapping` abstract class is defined in the abstract base classes module `collections.abc`. In other words, your `DynamicScope` class will inherit from the `abc.Mapping` abstract base class:

```
from collections import abc
class DynamicScope(abc.Mapping):
  ...
```

For more information about the `abc.Mapping` abstract base class, see the Python **documentation (https://docs.python.org/3/library/collections.abc.html#collections-abstract-base-classes)** .

Your `DynamicScope` class must support associative lookup operations (in other words, it must work like a Python dictionary). The "keys" in your faux dictionary will be the names of the variables in the referencing environment. The values for those "keys" will be the value of the variables with that name.

For example,

```
from dynamic_scope import get_dynamic_re
from typing import Any


def outer():
    a = "outer_a"
    b = "outer_b"
    c = "outer_c"
    return get_dynamic_re()

if __name__ == "__main__":
    dre = outer()
    print(f"{dre['a']=}")
    print(f"{dre['b']=}")
    print(f"{dre['c']=}")
```

will print

```
dre['a']='outer_a'
dre['b']='outer_b'
dre['c']='outer_c'
```

Abstract base classes are a **relatively new concept (https://peps.python.org/pep-3119/)** in Python. If you would like to work through some documentation and examples on implementing the `abc.Mapping` abstract base class in Python, I highly recommend reading **Becoming Noah Webster: Making Your Own Python Dictionary (https://uc.instructure.com/courses/1559602/pages/becoming-noah-webster-making-your-own-python-dictionary)**.

If a user of the `DynamicScope` class indexes a `DynamicScope` object using a variable name that is not bound in the object, it must raise a `NameError` exception. As an example:

```
variable_a = "This is variable a"
ds = DynamicScope()
ds["variable_a"] = variable_a
print(f"{ds['variable_a']=}")
print(f"{ds['variable_b']=}")
```

would print

```
ds['variable_a']='This is variable a'
Traceback (most recent call last):
  File "/home/hawkinsw/code/uc/cs3003-work/procedural/referencing_environment/example.py", line 97, in
    print(f"{ds['variable_b']=}")
  File "/home/hawkinsw/code/uc/cs3003-work/procedural/referencing_environment/dynamic_scope/__init__.py",
line 13, in __getitem__
    raise NameError(f"Name '{variable_name}' is not defined.")
NameError: Name 'variable_b' is not defined.
```

The `get_dynamic_re` Function

The `get_dynamic_re` function will populate a `DynamicScope` object and return that object with the names of the variables in the referencing environment as keys and the variable's values as values as bound using dynamic scoping.

Just how should we do that? Fortunately Python gives us some functionality for runtime inspection of the program. Not surprisingly that functionality can be found in the Python `inspect` (https://docs.python.org/3/library/inspect.html) module. You will rely heavily on the documentation of that module to complete this assignment.

The `inspect` module gives you the ability to get an array of **named tuples** (https://docs.python.org/3/glossary.html#term-named-tuple) that contain information about the *active* functions in the currently executing program by calling the `inspect.stack` (https://docs.python.org/3/library/inspect.html#inspect.stack) method.

To restate, the `inspect.stack` (https://docs.python.org/3/library/inspect.html#inspect.stack) method returns an array. Each element in that returned array is a named tuple of information about one of the active functions. Yes, I know, that's confusing! It'll be clearer with an example:

```
g_var = "Global variable"
def x():
  print(g_var)
  y()
def y():
  y_var = "Local variable in y()"
  z()
def z():
  __here__
  pass
x()
```

If you called `inspect.stack` at `__here__`, the resulting array would contain the following named tuples at the given indexes:

0. The information about the state of the function `y` when it called `z`.
1. The information about the state of the function `x` when it called `y`.
2. The information about program state when `x` started.

The first element of the named tuple that comprises each element in the list is a `Frame` object and you will work with these `Frame` objects to do most of the work in this lab. The documentation for `Frame`s can be found **online** (https://docs.python.org/3/reference/datamodel.html) and **here** (https://docs.python.org/3/library/inspect.html#types-and-members). Of particular interest for your work, look at the `f_locals` member and the `f_code` member.

Okay, I get why `f_locals` might be interesting, but why would we need access to the function's *code object*? Well, I'll tell you!

`f_locals` contains the (lexically-scoped-determined) values of all local *and* free variables in a function! What is a *free* variable? We will learn about the theoretical reason why Python uses the name *free* variables to describe certain variables, but the Python definition can be found in its documentation:

> If a variable is used in a code block but not defined there, it is a free variable.

More on the reason why this distinction is important after we talk about the algorithm for constructing a reference environment according to dynamic scoping:

1. Get a list of information about each of the executing functions, $stack\_info$.
2. Look at the `Frame` object, f of each of the tuples in $stack\_info$ from innermost to outermost.
    1. Add each of the local variables from f to the referencing environment, if a variable with that name does not already exist in the referencing environment.
    2. Make sure not to include *free* variables when performing this augmentation of the referencing environment!

We can get $stack\_info$ using the `inspect.stack()` method. We can get the names of the local variables from f from the keys of its `f_locals` member. We can get the values of local variables by indexing the `f_locals` member as if it were a dictionary (because, well, it is!) We can filter out the free variables by using the `f_code.co_freevars` member of f. `f_code.co_freevars` is a `tuple` but you can convert it to a `list` by wrapping it in a call to the `list()` function:

```
list(f.f_code.co_freevars)
```

So, let's return to the reason why we are filtering out *free* variables in our algorithm. Free variables do not actually give a value to a variable. The presence of a variable in the list of free variables of a function is just an indication that its value is accessed in that function. However, Python "helpfully" fills the values for those free variables in the `f_locals` dictionary for each frame. Just how does Python determine that value? Through static scoping, of course! Therefore, if we relied on the values in `f_locals` for free variables as we calculated our reference environment, then the contents would reflect static scoping and not dynamic scoping.

Be careful

When you call `inspect.stack()` from your `get_dynamic_re` function, the first named tuple of the list returned by `inspect.stack()` will contain information about the `get_dynamic_re` function itself! It's unintuitive, but it does make sense if you think about it! However, we will want to ignore that because we are concerned about the referencing environment of the statement *that called* `get_dynamic_re`!

Check Yourself Before You Wreck Yourself

And just how will a correct implementation work? A great question.

```
from dynamic_scope import get_dynamic_re
from typing import Any

def outer():
  a = "outer_a"
  b = "outer_b"
  c = "outer_c"
  d = "outer_d"
  e = "outer_e"

  def inner1():
    a = "inner1_a"
    b = "inner1_b"
    inner3("parameter_d")

  def inner2():
    a = "inner2_a"
    b = "inner2_b"
    inner3("parameter_d")

  def inner3(d: Any):
    e = "inner3_e"
    print(f"statically scoped value of a: {a}")
    print(f"statically scoped value of b: {b}")
    print(f"statically scoped value of c: {c}")
    print(f"statically scoped value of d: {d}")
    print(f"statically scoped value of e: {e}")
    dre = get_dynamic_re()
    print(f"dynamically scoped value of a: {dre['a']}")
    print(f"dynamically scoped value of b: {dre['b']}")
    print(f"dynamically scoped value of c: {dre['c']}")
    print(f"dynamically scoped value of d: {dre['d']}")
    print(f"dynamically scoped value of e: {dre['e']}")

  print("Calling inner1:")
  inner1()

  print("Calling inner2:")
  inner2()

if __name__ == "__main__":
  outer()
```

will print

```
Calling inner1:
statically scoped value of a: outer_a
statically scoped value of b: outer_b
statically scoped value of c: outer_c
statically scoped value of d: parameter_d
statically scoped value of e: inner3_e
dynamically scoped value of a: inner1_a
dynamically scoped value of b: inner1_b
dynamically scoped value of c: outer_c
dynamically scoped value of d: parameter_d
dynamically scoped value of e: inner3_e
Calling inner2:
statically scoped value of a: outer_a
statically scoped value of b: outer_b
statically scoped value of c: outer_c
statically scoped value of d: parameter_d
statically scoped value of e: inner3_e
dynamically scoped value of a: inner2_a
dynamically scoped value of b: inner2_b
dynamically scoped value of c: outer_c
```

```
dynamically scoped value of d: parameter_d
dynamically scoped value of e: inner3_e
```

That's a wall of text, right? Don't worry! The skeleton code distributed with this lab contains the same example, with explanatory comments!

Getting Started with the Assignment's Programming Environment

This assignment comes with skeleton code that will prepare you for successfully meeting all the criteria specified above. Download it **here (https://uc.instructure.com/courses/1559602/files/158896109?wrap=1)** (https://uc.instructure.com/courses/1559602/files/158896109/download?download_frd=1) .

`dynamic_scope/__init__.py`

Your implementation of the `DynamicScope` class and the `get_dynamic_re` function will go in the `dynamic_scope/__init__.py` file. The file/directory structure turns your code into a Python **module (https://docs.python.org/3/tutorial/modules.html)** -- cool! As a result, you can use your `DynamicScope` class and `get_dynamic_re` function in any other Python program by

```
from dynamic_scope import get_dynamic_re
```

`skeleton_test.py`

`skeleton_test.py` contains code that `import`s your `dynamic_scope` module and runs unit tests on the implementations of the `DynamicScope` class and the `get_dynamic_re` function. If you have correctly implemented the `DynamicScope` class and the `get_dynamic_re` function according to the criteria described herein, running the code in that file should produce the following output:

```
result=<unittest.result.TestResult run=0 errors=0 failures=0>
```

Tips and Tricks

In the directory structure where you are doing your development, you may want to create a *virtual environment*. You can read more about those **here (https://packaging.python.org/en/latest/guides/installing-using-pip-and-virtual-environments/#creating-a-virtual-environment)** .

No matter whether you are a Windows- or macOS-based developer (and more power to you if you are a Linux-based developer!), you may want to consider using Codium (or the non-free Visual Studio Code) to develop the code for this assignment. More information about doing Python

development in those IDEs is available **online** (https://code.visualstudio.com/docs/python/python-tutorial) .

Finally, I recommend writing a small Python program that you can use for experimentation. Use `print` function calls, `__dict__` attributes, `for` loops, etc, to work through the list generated by `inspect.stack`, the `f_locals` and the `f_locals.co_freevars`. The more you play around and explore these different variables and their contents, the easier this assignment will be! Do not attempt to implement the entire algorithm using only the test cases as your method of feedback.

Extra Credit

There is an opportunity for you to earn 5 extra credit points on this assignment.

Consider the following Python code:

```
def late_local():
  def outer():
    a = "outer_a"
    def inner():
      print(f"{a=}")
      a = "inner_a"
    inner()
  outer()
late_local()
```

The

```
      a = "inner_a"
```

in `inner()` creates a local variable named `a` but the

```
      print(f"{a=}")
```

references that variable before it is given an initial value. As I am sure you are aware, Python does not allow this:

```
Traceback (most recent call last):
  File "check.py", line 27, in <module>
    late_local()
  File "check.py", line 25, in late_local
    outer()
  File "check.py", line 24, in outer
    inner()
  File "check.py", line 22, in inner
    print(f"{a=}")
UnboundLocalError: local variable 'a' referenced before assignment
```

For an additional 5 points of credit on this assignment,

```
def test_late_local():
  def outer():
    a = "outer_a"
```

```
    def inner():
      dre = get_dynamic_re()
      a = "inner_a"
      return dre
    return inner()
  return outer()
dre = test_late_local()
print(f"{dre['a']=}")
```

should raise `UnboundLocalError`. If you are wondering where/how to get started, I recommend looking inside the `Frame` object's `f_code` attribute. That will point you to a so-called code object which has several very interesting attributes. Of those attributes, the union of `co_varnames` and `co_cellvars` contain all the local variables, even the ones that have not been given values. On the other hand, the `f_locals` that you worked with above, contain the local variables that have values. You can see more about that online at the documentation for the `inspect` **module (https://docs.python.org/3/library/inspect.html)** .

Submitting Your Work

Submitting your work could not be easier! For this lab you will know, without a doubt, your score on 90% of the points available. Thanks to an autograder, you will be able submit, resubmit and resubmit again until your implementation is correct! It's pretty cool.

Associated with the **Canvas Assignment (https://uc.instructure.com/courses/1559602/assignments/19305142)** for this assignment is a link to Gradescope. When you open Gradescope you will see a link for Assignment 1 - Dynamo of Volition. Follow the instructions for uploading your submission.

The only file that you will submit for this assignment is `__init__.py` from the `dynamic_scope` directory. Every time that you submit, your code will be run against automated tests. ***Important***: Your submission must be named `__init__.py` -- not, `__Init__.py` or `Init____.py` or even `__INIT__.py`. The autograder is *very* temperamental. In that way, it is like me! The autograder will tell you exactly how many points (out of 90) you earned.

You have *unlimited* submission opportunities until the assignment deadline to score all 90 points. The final ten points will be awarded based on

Criteria	Points	Description
Code Quality	10	Code must 1. be well documented, 2. use meaningful variable names, 3. use consistent styling, and 4. generally be professional. Code that largely meets these criteria will receive 10 points. Code that mostly meets these criteria will receive 5 points. Code that does not meet these criteria will receive 0 points.