CptS 111, Spring 2023
Lect. #13, Mar. 1, 2023
Class Notes

Today's Agenda:

1. Namespaces, scope, and scope resolution
2. Global variables

Ch. 5 (cont.)

I mentioned previously that we can use the same names inside a function as we do outside the function, and Python will be able to tell the difference. This has to do with the following concepts:

1. Namespaces, Scope, and Scope Resolution

These concepts can be a little confusing, so let's first look at some examples.

In [1]:
```
# Variable local to the function

def increment_x(x):
    x = x + 1
    print('This is the value of x inside the function:', x)
    return
x = 42
increment_x(x)
print('This is the value of x outside the function:', x)
```

```
This is the value of x inside the function: 43
This is the value of x outside the function: 42
```

As you can see from the example above, `x` didn't change outside the function even though we called the function and added 1 to `x` before printing `x` outside the function. ***This is because `x` inside the function is local to the function, i.e., it's a local variable.***

In [2]:
```
# Another example

def print_pi():
    pi = 3.141592653
    print('This is the value of pi inside the function:', pi)
    return
pi = 3.141
print_pi()
print('This is the value of pi outside the function:', pi)
```

```
This is the value of pi inside the function: 3.141592653
This is the value of pi outside the function: 3.141
```

In the example above, the value of `pi` is different inside and outside the function. Again, this is because `pi` inside the function is local to the function.

In [3]:
```
# An example when a name inside the function isn't local to it

def circ_circum(r):
    circum = 2 * pi * r
    print(f'Circle circumference inside the function: {circum:g}')
    return circum
pi = 3.141592653
rad = 2
print(f'Circle circumference outside the function: {circ_circum(rad):g}')
```

```
Circle circumference inside the function: 12.5664
Circle circumference outside the function: 12.5664
```

In the example above, the function `circ_circum()` found the value of `pi` even though `pi` is outside the function. We'll discuss this very soon!

Next, let's turn to something that seems irrelevant at this point. We'll use a function as a name for an lvalue.

In [4]:
```
# Using a function as a name for an lvalue

int = 42
print(int)
type(int)
```

```
42
```

Out[4]:
```
int
```

In the example above, we've assigned `42` to the function `int()`, and we see `int`'s type is now `int`. What happens when we try to use `int()` now to convert the string value result of the `input()` function?

In [5]:
```
# Using int() after we've used int as an lvalue

integer = int(input('Enter an integer: '))
print(integer)
```

```
Enter an integer: 42

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [5], in <cell line: 3>()
      1 # Using int() after we've used int as an lvalue
----> 3 integer = int(input('Enter an integer: '))
      4 print(integer)

TypeError: 'int' object is not callable
```

We end up with an error because the Python interpreter finds the `int` we've defined and, thus, doesn't look into the built-in namespace. Next consider the following. Hang on because we'll discuss all this very soon!

In [6]:
```
# Using int() inside a function after we've used int as an lvalue

def get_input():
    integer = input('Enter an integer: ')
    integer = int(integer)
    return integer

int_num = get_input()
```

```
Enter an integer: 42

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [6], in <cell line: 8>()
      5     integer = int(integer)
      6     return integer
----> 8 int_num = get_input()

Input In [6], in get_input()
      3 def get_input():
      4     integer = input('Enter an integer: ')
----> 5     integer = int(integer)
      6     return integer

TypeError: 'int' object is not callable
```

The Python interpreter doesn't find `int()` locally so it looks globally and finds our definition of `int`.

So how does this all work? Before we get to the answer, we need to define some terms:

- *scope*: area(s) of code where name is visible
- *namespace*: mapping of all defined names to their objects (values, functions, and so on)
- *scope resolution*: process of searching possible namespaces for names

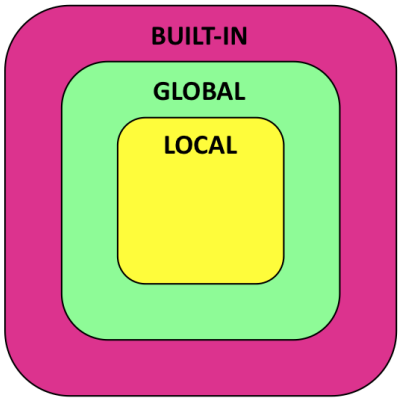
There are three namespaces: local, global, and built-in. **Python *always* searches them in the same order:**

```
local -> global -> built-in
```

What are these three different namespaces? Well, we can think of them in terms of their *scope*, i.e., where their names are visible in a program

- local scope: visible within a function
- global scope: visible everywhere when defined
- built-in scope: visible everywhere and always present (e.g., `print()`, `input()`)

You can picture namespace scope and scope resolution as nested areas. Python first looks in the local namespace, then in the global namespace, then in the built-in namespace, and this is exactly why we obtained the results in the examples above.



- Variables assigned in a function are in *local namespace* and, thus, only have local scope, i.e., Python won't find these outside the function.
- Variables assigned outside a function are in *global namespace* and, thus, if they're used inside a function, Python will first search for the variable in local namespace (inside the function) and then it will look in global namespace.
- If Python can't find a variable inside the function (in local namespace) or outside the function (in global namespace), it will look for a built-in function with the name in *built-in namespace*.

Let's consider one more example:

In [7]:
```
# Use of the same name inside and outside a function

def circ_area():
    area = pi * r ** 2
    return area
pi = 3.141592
r = float(input('Enter radius: '))
area = circ_area()
print(f'Circle area: {area:g}')
```

```
Enter radius: 3
Circle area: 28.2743
```

In this example, both `pi` and `r` are global variables. Note, however, that the lvalue named `area` assigned inside the function is a local variable. It has to be returned to the calling function because it has no scope outside the function. OTOH, the lvalues `area` both inside and outside the function don't confuse Python. We could have used different names for each, but it's fine to use the same name.

Note that it's usually better to pass arguments as parameters to a function.

There's one last point we should consider when it comes to using names with functions. Consider the following example.

In [8]:
```
# Use of list as argument in a function call

def add_float(floats):
    num = float(input('Enter a decimal number: '))
    floats.append(num)
    return                    # Void function; nothing returned

float_list = [1.1, 2.2, 3.3]
add_float(float_list)
print(f'float_list is: {float_list}.')
```

```
Enter a decimal number: 4.4
float_list is: [1.1, 2.2, 3.3, 4.4].
```

How did this happen? We called the void function `add_float()` with the list argument `float_list`, which was then assigned to the list parameter `floats`. Then we added a float to `floats`, but we didn't return `floats`. How then did `float_list` change? **When `float_list` is passed to the function `add_float()`, the parameter `floats` points to the same location in memory as `float_list` and, as such, it has the same "value." Because `float_list` is a list, and lists are mutable, when a change is made to `floats`, the "value" is changed in memory. `float_list` still points to this location in memory, so it now has a new "value."**

Let's look at zPA 5.8.2

2. Global variables

If you follow the rules below, the `global` command should be straightforward to understand.

1. Use global variables sparingly. **If you need to use something in a function, then pass it as an argument.**
2. If you need to use global variables in a function, try to put them all in one location and identify them as global variables.
3. *Use the `global` command when you need to **modify** a value inside the function.*

Let's look at an example when you need to use `global`. Suppose you want to increment an accumulator in a function that you call more than once. Then you'll need to initialize it outside the function.

In [9]:
```
# Silly function to show when you need to initialize an accumulator
# outside the function and use the global command inside the function.

def inc_accumulator():
    i = 0
    i += 1
    print(f'This function has been called {i} time(s).')
    return

inc_accumulator()   # Call function once
inc_accumulator()   # Call function twice
```

```
This function has been called 1 time(s).
This function has been called 1 time(s).
```

The accumulator `i` is reset to 0 each time the function is called, so `i` never increases regardless of how many times the function is called!

In [10]:
```
# Still not correct silly function.

i = 0
def inc_accumulator_v2():
    i = i + 1
    print(f'This function has been called {i} time(s).')
    return

inc_accumulator_v2()
inc_accumulator_v2()
```

```
---------------------------------------------------------------------------
----
UnboundLocalError                         Traceback (most recent call l
ast)
Input In [10], in <cell line: 9>()
      6     print(f'This function has been called {i} time(s).')
      7     return
----> 9 inc_accumulator_v2()
     10 inc_accumulator_v2()

Input In [10], in inc_accumulator_v2()
      4 def inc_accumulator_v2():
----> 5     i = i + 1
      6     print(f'This function has been called {i} time(s).')
      7     return

UnboundLocalError: local variable 'i' referenced before assignment
```

Now we get an error because `i` wasn't defined, i.e., `i += 1` tries to add 1 to `i`, but `i` wasn't defined!

In [11]:
```
# Correct version of silly function.

i = 0  # global variable
def inc_accumulator_v3():
    global i                  # Use global command the way we did in PA #4
    i = i + 1
    print(f'This function has been called {i} time(s).')
    return

inc_accumulator_v3()
inc_accumulator_v3()
inc_accumulator_v3()
```

```
This function has been called 1 time(s).
This function has been called 2 time(s).
This function has been called 3 time(s).
```

This function works correctly because we initialized `i` outside the function, but then we declared it to be global inside the function, so when the function couldn't find it locally, it looked for it globally.