PDSP 2025, Lecture 11, 11 September 2025

Scope and global variables

- The scope of a variable refers to the portion of the program where its value is available
- If we refer to a value that is not defined in a function, it is looked up in the global context

In [1]:
```
def f():
    y = x + 22
    print(y)
    return
```

In [2]:
```
x = 7
f()
```

```
29
```

- As soon as we assign a variable a value inside a function, *all* instances of that variable are treated as local to the function
- This decision is *static* based on the program text. In the code below, we cannot be sure that the assignment `x = 33` will execute, but Python still denotes `x` to be local to `f()`
- Though the check is based on the static program text, the error is flagged only when the function executes. The definition of `f()` does not trigger an error though the problem is evident in the text of the function.

In [3]:
```
def f():
    y = x + 22
    print(y)
    if y > 1000:
        x = 33
    return
```

In [4]:
```
x = 7
f()
```

```
---------------------------------------------------------------------------
-
UnboundLocalError                         Traceback (most recent call las
t)
Cell In[4], line 2
      1 x = 7
----> 2 f()

Cell In[3], line 2, in f()
      1 def f():
----> 2     y = x + 22
      3     print(y)
      4     if y > 1000:

UnboundLocalError: cannot access local variable 'x' where it is not associ
ated with a value
```

- This static check applies even if it is impossible for the local assignment to be executed

In [5]:
```
def checky():
    y = x + 2
    return
    if False:
        x = 7
```

In [6]:
```
x = 8
checky()
```

```
---------------------------------------------------------------------------
-
UnboundLocalError                         Traceback (most recent call las
t)
Cell In[6], line 2
      1 x = 8
----> 2 checky()

Cell In[5], line 2, in checky()
      1 def checky():
----> 2     y = x + 2
      3     return
      4     if False:

UnboundLocalError: cannot access local variable 'x' where it is not associ
ated with a value
```

- More examples of using global values within a function without redefining the variable

In [7]:
```
def display_count():
    print(count)
    return
```

In [8]:
```
def display_upto_count():
    for i in range(count):
```

```
        print(count+i)
    return
```

- If we call `display_count()` without a global definition for `count` we get an error

In [9]:
```
display_count()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 display_count()

Cell In[7], line 2, in display_count()
      1 def display_count():
----> 2     print(count)
      3     return

NameError: name 'count' is not defined
```

- If `count` is available in the global context, the two functions work as expected

In [10]:
```
count = 7
```

In [11]:
```
display_count()
```

```
7
```

In [12]:
```
display_upto_count()
```

```
7
8
9
10
11
12
13
```

- If we try to update `count` inside the function, both occurrences become local
- The occurrence on the right hand side of the assignment generates an error because its value is now undefined
- Once again, this *static* error is only triggered at run-time when the function executes

In [13]:
```
def increment_local(k):
    count = count+k
    return
```

In [14]:
```
increment_local(2)
```

```
---------------------------------------------------------------------------
-
UnboundLocalError                         Traceback (most recent call las
t)
Cell In[14], line 1
----> 1 increment_local(2)

Cell In[13], line 2, in increment_local(k)
      1 def increment_local(k):
----> 2     count = count+k
      3     return

UnboundLocalError: cannot access local variable 'count' where it is not as
sociated with a value
```

- Reassigning a variable within a function disconnects it from the external variable with the same name

In [15]:
```
def reset_local(k):
    count = k
    return
```

In [16]:
```
reset_local(77)
```

In [17]:
```
count
```

Out[17]: 7

- We can declare a variable to be `global` to override Python's default scope rules
- `global` tells Python to treat the variable inside the function as one from the global context

In [18]:
```
def increment_global(k):
    global count
    count = count+k
    return
```

In [19]:
```
increment_global(8)
```

In [20]:
```
display_count()
```

```
15
```

- The default rule about local scope applies to mutable values as well

In [21]:
```
def concat_local():
    l1 = l1 + l2
    return
```

In [22]:
```
l1 = [1,2,3]
l2 = [4,5,6]
concat_local()
```

```
---------------------------------------------------------------------------
-
UnboundLocalError                         Traceback (most recent call las
t)
Cell In[22], line 3
      1 l1 = [1,2,3]
      2 l2 = [4,5,6]
----> 3 concat_local()

Cell In[21], line 2, in concat_local()
      1 def concat_local():
----> 2     l1 = l1 + l2
      3     return

UnboundLocalError: cannot access local variable 'l1' where it is not assoc
iated with a value
```

In [23]:
```
def concat_global():
    global l1
    l1 = l1 + l2
    return
```

In [24]:
```
l1 = [1,2,3]
l2 = [4,5,6]
concat_global()
```

In [25]:
```
l1, l2
```

Out[25]:
```
([1, 2, 3, 4, 5, 6], [4, 5, 6])
```

- We can define a value inside a function and "export" it outside by declaring it `global`

In [26]:
```
del(l1)
del(l2)
```

In [27]:
```
def concat_global():
    global l1
    l1 = [1,2,3]
    l1 = l1 + l2
    return
```

In [28]:
```
l2 = [4,5,6]
concat_global()
```

In [29]:
```
l1
```

Out[29]:
```
[1, 2, 3, 4, 5, 6]
```

- The following would work with *dynamic* scoping -- based on execution of program
  - `init()` defines `b` and then calls `seta()`, so with dynamic scoping, `b` is known to `seta()`

- Python uses *static* scoping -- based on text of program -- so this code generates an error
- Most languages use static scoping because dynamic scoping makes it hard to reason about correctness

In [30]:
```
def seta():
    a = b + 5
    print(a)

def init():
    b = 7
    seta()

init()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[30], line 9
      6     b = 7
      7     seta()
----> 9 init()

Cell In[30], line 7, in init()
      5 def init():
      6     b = 7
----> 7     seta()

Cell In[30], line 2, in seta()
      1 def seta():
----> 2     a = b + 5
      3     print(a)

NameError: name 'b' is not defined
```