Lecture 09, 12 September 2024

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

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()`

In [2]:
```
def f():
    y = x + 22
    print(y)
    if y > 1000:
        x = 33
    return

x = 7
f()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[2], line 9
      6     return
      8 x = 7
----> 9 f()

Cell In[2], 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 associated with a value
```

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

In [3]:
```
def display_count():
    print(count)
    return
```

In [4]:
```
def display_upto_count():
    for i in range(count):
        print(count+i)
    return
```

In [5]:
```
count = 7
```

In [6]:
```
display_count()
```

```
7
```

In [7]:
```
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

In [8]:
```
def increment_local(k):
    count = count+k
    return
```

In [9]:
```
increment_local(2)
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[9], line 1
----> 1 increment_local(2)

Cell In[8], 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 associated with a value
```

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

In [10]:
```
def reset_local(k):
    count = k
    return
```

In [11]:
```
reset_local(77)
```

In [12]:
```
count
```

Out[12]: 7

- We can declare a variable to be `global` to override Python's default scope rules

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

In [14]:
```
increment_global(8)
```

In [15]:
```
display_count()
```

```
15
```

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

In [16]:
```
def concat_local():
    l1 = l1 + l2
    return
```

In [17]:
```
l1 = [1,2,3]
l2 = [4,5,6]
concat_local()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[17], line 3
      1 l1 = [1,2,3]
      2 l2 = [4,5,6]
----> 3 concat_local()

Cell In[16], 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 associated with a value
```

In [18]:
```
def concat_global():
    global l1
    l1 = l1 + l2
    return
```

In [19]:
```
l1 = [1,2,3]
l2 = [4,5,6]
concat_global()
```

In [20]:
```
l1, l2
```

Out[20]: ([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 [21]:
```
del(l1)
del(l2)
```

In [22]:
```
def concat_global():
    global l1
    l1 = [1,2,3]
    l1 = l1 + l2
    return
```

In [23]:
```
l2 = [4,5,6]
concat_global()
```

In [24]:
```
l1
```

Out[24]:
```
[1, 2, 3, 4, 5, 6]
```