PDSP 2025, Lecture 10, 9 September 2025

Mutable and immutable values

- Lists and dictionaries are mutable
- `int`, `float`, `bool`, `str`, tuple are immutable

- For immutable values, assignment copies the value

In [1]:
```
x = 5
y = x
y = 7  # Does not affect the value of x
```

In [2]:
```
x,y
```

Out[2]:
```
(5, 7)
```

- For mutable values, assigment *aliases* the new name to point to the same value as the old name
- Updating through either name affects both

In [3]:
```
l1 = [1,2,3]
l2 = l1
l2[0] = 4
```

In [4]:
```
l1,l2
```

Out[4]:
```
([4, 2, 3], [4, 2, 3])
```

In [5]:
```
l1[2] = 6
```

In [6]:
```
l1,l2
```

Out[6]:
```
([4, 2, 6], [4, 2, 6])
```

- We can update a mutable value inside a function
- However, we should be careful to use updates that do not reassign the name
- Use `l.append(v)` vs `l = l + [v]`

In [7]:
```
def bad(l,v):
    l = l + [v]
    print(l)
    return
```

In [8]:
```
def good(l,v):
    l.append(v)
    print(l)
    return
```

- `bad(l,v)` appends `v` within the function, but creates a new copy of `l` in the process, that is different from the `l` passed as an argument

In [9]:
```
l = [1,2,3]
```

In [10]:
```
bad(l,4)
```

```
[1, 2, 3, 4]
```

In [11]:
```
l
```

Out[11]:
```
[1, 2, 3]
```

`good(l,v)` on the other hand updates `l` in place, so the effect is visible outside

In [12]:
```
l
```

Out[12]:
```
[1, 2, 3]
```

In [13]:
```
good(l,4)
```

```
[1, 2, 3, 4]
```

In [14]:
```
l
```

Out[14]:
```
[1, 2, 3, 4]
```

- We can update `bad(l,v)` to return the modified list, but then we have to reassign `l` to the returned value

In [15]:
```
def bad2(l,v):
    l = l + [v]
    print(l)
    return(l)
```

In [16]:
```
l = [1,2,3]
returnlist = bad2(l,4)
```

```
[1, 2, 3, 4]
```

In [17]:
```
l,returnlist
```

Out[17]:
```
([1, 2, 3], [1, 2, 3, 4])
```

In [18]:
```
l = [1,2,3]
l = bad2(l,4)
```

```
[1, 2, 3, 4]
```

In [19]:
```
l
```

Out[19]:
```
[1, 2, 3, 4]
```

Slices and copying lists

- A slice creates a new list
- Full slice `l[:]` is a faithful copy of `l`
  - Abbreviation for `l[0:len(l)]`
- Assigning a full slice makes a disjoint copy of a list

In [20]:
```
l1 = [1,2,3]
l2 = l1[:]
```

In [21]:
```
l1,l2
```

Out[21]:
```
([1, 2, 3], [1, 2, 3])
```

In [22]:
```
l1[2] = 6
l2[0] = 4
```

In [23]:
```
l1, l2
```

Out[23]:
```
([1, 2, 6], [4, 2, 3])
```

Pitfalls of mutability

In [24]:
```
zerorow = [0,0,0]
zeromat = [zerorow, zerorow, zerorow]
```

In [25]:
```
zeromat
```

Out[25]:
```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

In [26]:
```
zeromat[2][2] = 33
```

In [27]:
```
zeromat
```

Out[27]:
```
[[0, 0, 33], [0, 0, 33], [0, 0, 33]]
```

In [28]:
```
zerorow
```

Out[28]:
```
[0, 0, 33]
```

- This happens because updating any row in `zeromat` impliciltly updates `zerolist`
- And vice versa

In [29]:
```
zerorow[0] = 11
```

In [30]:
```
zeromat
```

Out[30]:
```
[[11, 0, 33], [11, 0, 33], [11, 0, 33]]
```

An aside

- Multiplication is repeated addtion: $n \times m = \underbrace{n + n + \cdots + n}_{\text{m-times}}$
- For lists, `+` denotes concatenation
- `l+l+l+l` can be written as `l*4`

In [31]:
```
4 + 4 + 4
```

Out[31]:
```
12
```

In [32]:
```
4*3
```

Out[32]:
```
12
```

In [33]:
```
[0,0,0] + [0,0,0] + [0,0,0]
```

Out[33]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0]
```

In [34]:
```
[0,0,0]*3
```

Out[34]:
```
[0, 0, 0, 0, 0, 0, 0, 0, 0]
```

- This does not avoid list aliasing issues

In [35]:
```
zerorow = [0,0,0]
```

In [36]:
```
zerolist = [zerorow]*3
```

In [37]:
```
zerolist
```

Out[37]:
```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

In [38]:
```
zerolist[1][1] = 44
```

In [39]:
```
zerolist
```

Out[39]:
```
[[0, 44, 0], [0, 44, 0], [0, 44, 0]]
```

- Use list comprehension instead
- Each list comprehension creates a new list

In [40]:
```
[ 0 for i in range(3) ]  # A list of 3 zeros
```

Out[40]: [0, 0, 0]

In [41]:
```
[ [ 0 for i in range(3) ] for j in range (3) ] # 3 disjoint lists of 3 zeros
```

Out[41]: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

In [42]:
```
zmat = [ [ 0 for i in range(3) ] for j in range (3) ]
```

In [43]:
```
zmat
```

Out[43]: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

In [44]:
```
zmat[1][1] = 1
```

In [45]:
```
zmat
```

Out[45]: [[0, 0, 0], [0, 1, 0], [0, 0, 0]]

Calling functions

- Suppose we have a function definition `def f(a,b):` and a function call `f(x,y)`
- When `f(x,y)` is executed, it is as though we start `f` with the assignments

```
a = x
b = y
```

- This explains how/when values can be updated within a function

In [46]:
```
def factorial(n):
    ans = 1
    while n >= 1:
        ans = ans * n
        n = n-1
    return(ans)
```

In [47]:
```
x = 6
y = factorial(x)
```

In [48]:
```
x,y
```

Out[48]: (6, 720)

- Inside the function, the parameter `n` is decremented to `0`
- `n` is derived from the variable `x` passed when the function is called
- Since `x` is immutable, the implicit assignment `n = x` *copies* the value of `x` into `n`
- Updating `n` has no effect on `x`

- This also means we cannot write a function `swap` along the following lines

In [49]:
```
def swap(x,y):
    (x,y) = (y,x)
    return
```

In [50]:
```
m = 5
n = 7
swap(m,n)
```

In [51]:
```
m,n
```

Out[51]: (5, 7)

- This will not work with mutable values either
- The problem is the reassignment inside the function

In [52]:
```
list1 = [1,2,3]
list2 = [4,5,6]
swap(list1,list2)
```

In [53]:
```
list1, list2
```

Out[53]:
```
([1, 2, 3], [4, 5, 6])
```

Passing mutable values to a function

- Passing an argument is like executing an assignment statement before starting the function
- For mutable values, this aliases the function parameter to the called value
- In place changes in the function affect the value outside the function

In [54]:
```
def concat(l1,l2):
    l1.extend(l2)
    return
```

In [55]:
```
l3 = [1,2,3]
l4 = [4,5,6]
concat(l3,l4)
```

In [56]:
```
l3,l4
```

Out[56]:
```
([1, 2, 3, 4, 5, 6], [4, 5, 6])
```

- If we pass a slice, the value in the function is a disjoint copy

In [57]:
```
l3 = [1,2,3]
l4 = [4,5,6]
concat(l3[:],l4[:])
```

In [58]:
```
l3,l4
```

Out[58]:
```
([1, 2, 3], [4, 5, 6])
```

- However, reassigning the variable inside the function creates a new value not connected to the outer value

In [59]:
```
def concat2(l1,l2):
    l1 = l1 + l2
    return
```

In [60]:
```
l3 = [1,2,3]
l4 = [4,5,6]
concat2(l3,l4)
```

In [61]:
```
l3,l4  # No effect - reassignment in function creates a local copy
```

Out[61]:
```
([1, 2, 3], [4, 5, 6])
```

- In fact, our problem with `swap()` applies to mutable values as well
- The statement `(m,n) = (n,m)` is a reassignment and creates new values inside the function

In [62]:
```
swap(l3,l4)
```

In [63]:
```
l3,l4
```

Out[63]:
```
([1, 2, 3], [4, 5, 6])
```

- Be careful not to mix reassignment with in-place modification
- What is the outcome of the following?

In [64]:
```
def myappend(l,x):
    l = l.append(x)
    return(l)
```

In [65]:
```
l1 = [1,2]
l1 = myappend(l1,3)
```

In [66]:
```
l1
```

In [67]:
```
print(l1)
```

```
None
```

- `None` is a special value in Python that explicitly represents that no value is assigned
- A function that does not return a value returns `None`
- In the notebook, the value is "empty", but `print()` displays it as `None`
  - In other words, `str(None)` converts the value `None` to the string `"None"`
- `None` has its own type which is not compatible with any other type, so no operations are legal

In [68]:
```
str(None)
```

Out[68]:
```
'None'
```

In [69]:
```
print(None)
```

```
None
```

In [70]:
```
type(None)
```

Out[70]:
```
NoneType
```

- Setting a variable to `None` is different from leaving it undefined

In [71]:
```
x = 7
```

In [72]:
```
type(x)
```

Out[72]:
```
int
```

In [73]:
```
del(x)
```

In [74]:
```
x
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[74], line 1
----> 1 x

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

In [75]:
```
x = None
```

In [76]:
```
x
```

- We can test if a variable is set to `None`
- We will use this later

In [77]:
```
x == None
```

Out[77]:
```
True
```

More on equality

- `x == y` checks that `x` and `y` contain the same value
- An assignment `l2 = l1` aliases `l2` to point to the same list as `l1`
  - Naturally, we expect `l2 == l1` to be `True`

    - But there is a stronger relationship, because `l1` and `l2` are the *same* value
- `x is y` checks if `x` and `y` refer to the same value
    - If `x is y` holds, it must be that `x == y`
    - Converse is not true

In [78]:
```
l1 = [1,2,3]
l2 = l1
l3 = l1[:]
```

In [79]:
```
l1 == l2, l1 == l3
```

Out[79]: (True, True)

In [80]:
```
l1 is l2, l1 is l3
```

Out[80]: (True, False)

- `x is y` can also be tested for immutable values, but the outcome is not useful or reliable

In [81]:
```
x = 5
y = x
```

In [82]:
```
x is y # Not useful for immutable values
```

Out[82]: True

In [83]:
```
x = 5
y = 5
```

In [84]:
```
x is y
```

Out[84]: True

In [85]:
```
s = "hello"
t = s
```

In [86]:
```
s is t
```

Out[86]: True

In [87]:
```
s = "hello"
t = "hello"
```

In [88]:
```
s is t
```

Out[88]: True