Python Notional Machine

Our goal is to refresh ourselves on basics (and some subtleties) associated with Python's data and computational model. Along the way, we'll also use or refresh ourselves on the **environment model** as a way to think about and keep track of the effect of executing Python code. Specifically, we'll demonstrate use of *environment diagrams* to explain the outcomes of different code sequences.

Variables and data types

Integers

In [1]:
```
a = 307
b = a
print('a:', a, '\nb:', b)
```

```
a: 307
b: 307
```

In [2]:
```
a = a + 310
a += 400
print('a:', a, '\nb:', b)
```

```
a: 1017
b: 307
```

So far so good -- integers, and variables pointing to integers, are straightforward.

Lists

In [3]:
```
x = ['baz', 302, 303, 304]
print('x:', x)
```

```
x: ['baz', 302, 303, 304]
```

In [4]:
```
y = x
print('y:', y)
```

```
y: ['baz', 302, 303, 304]
```

In [5]:
```
x = 377
print('x:', x, '\ny:', y)
```

```
x: 377
y: ['baz', 302, 303, 304]
```

Unlike integers, lists are mutable:

In [6]:
```
x = y
x[0] = 388
print('x:', x)
```

```
x: [388, 302, 303, 304]
```

In [7]:
```
print('y:', y)
```

```
y: [388, 302, 303, 304]
```

As seen above, we have to be careful about sharing (also known as "aliasing") mutable data!

In [8]:
```
a = [301, 302, 303]
b = [a, a, a]
print(b)
```

```
[[301, 302, 303], [301, 302, 303], [301, 302, 303]]
```

In [9]:
```
b[0][0] = 304
print(b)
print(a)
```

```
[[304, 302, 303], [304, 302, 303], [304, 302, 303]]
[304, 302, 303]
```

Tuples

Tuples are a lot like lists, except that they are immutable.

In [10]:
```
x = ('baz', [301, 302], 303, 304)
y = x
print('x:', x, '\ny:', y)
```

```
x: ('baz', [301, 302], 303, 304)
y: ('baz', [301, 302], 303, 304)
```

Unlike a list, we can't change the top most structure of a tuple. What happens if we try the following?

In [11]:
```
x[0] = 388
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-11-8a08f6fbfa16> in <module>
----> 1 x[0] = 388

TypeError: 'tuple' object does not support item assignment
```

What will happen in the following (operating on `x`)?

In [12]:
```
x[1][0] = 311
print('x:', x, '\ny:', y)
```

```
x: ('baz', [311, 302], 303, 304)
y: ('baz', [311, 302], 303, 304)
```

So we still need to be careful! The tuple didn't change at the top level -- but it might have members that are themselves mutable.

Strings

Strings are also immutable. We can't change them once created.

In [13]:
```
a = 'ya'
b = a + 'rn'
print('a:', a, '\nb:', b)
```

```
a: ya
b: yarn
```

In [14]:
```
a[0] = 'Y'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-150fa00334bc> in <module>
----> 1 a[0] = 'Y'

TypeError: 'str' object does not support item assignment
```

In [15]:
```
c = 'twine'
d = c
c += ' thread'
print('c:', c, '\nd:', d)
```

```
c: twine thread
d: twine
```

That's a little bit tricky. Here the `+=` operator makes a copy of `c` first to use as part of the new string with `' there'` included at the end.

Back to lists: append, extend, and the '+' and '+=' operators

In [16]:
```
x = [1, 2, 3]
y = [4, 5]
x.append(y)
y[0] = 99
print('x:', x, '\ny:', y)
```

```
x: [1, 2, 3, [99, 5]]
y: [99, 5]
```

So again, we have to watch out for aliasing/sharing, whenever we mutate an object.

In [17]:
```
x = [1, 2, 3]
y = [4, 5]
x.extend(y)
y[0] = 88
print('x:', x, '\ny:', y)
```

```
x: [1, 2, 3, 4, 5]
y: [88, 5]
```

```
</pre>
What happens when using the  +  operator used on lists?
```

In [18]:
```
x = [1, 2, 3]
y = x
x = x + [4, 5]
print('x:', x)
```

```
x: [1, 2, 3, 4, 5]
```

So the `+` operator on a list looks sort of like extend. But has it changed `x` in place, or made a copy of `x` first for use in the longer list?

And what happens to `y` in the above?

In [19]:
```
print('y:', y)
```

```
y: [1, 2, 3]
```

So that clarifies things -- the `+` operator on a list makes a (shallow) copy of the left argument first, then uses that copy in the new larger list.

Another case, this time using the `+=` operator with a list. Note: in the case of integers, `a = a + <val>` and `a += <val>` gave exactly the same result. How about in the case of lists?

In [20]:
```
x = [1, 2, 3]
y = x
x += [4, 5]
y[0] = 77
print('x:', x, '\ny:', y)
```

```
x: [77, 2, 3, 4, 5]
```

```
y: [77, 2, 3, 4, 5]
```

So `x += <something>` is NOT the same thing as `x = x + <something>` if `x` is a list! Here it actually DOES mutate or change `x` in place, if that is allowed (i.e., if `x` is a mutable object).

Contrast this with the same thing, but for `x` in the case where `x` was a string. Since strings are immutable, python does not change `x` in place. Rather, the `+=` operator is overloaded to do a top-level copy of the target, make that copy part of the new larger object, and assign that new object to the variable.

Let's check your understanding. What will happen in the following, that looks just like the code above for lists, but instead using tuples. What will x and y be after executing this?

In [21]:
```
x = (301, 302, 303)
y = x
x += (304, 305)
print('x:', x, '\ny:', y)
```

```
x: (301, 302, 303, 304, 305)
y: (301, 302, 303)
```

Functions and scoping

In [22]:
```
x = 500
def foo(y):
    return x + y
z = foo(307)
print('x:', x, '\nfoo:', foo, '\nz:', z)
```

```
x: 500
foo: <function foo at 0x7faf84061940>
z: 807
```

In [23]:
```
def bar(x):
    x = 1000
    return foo(307)
w = bar('hi')
print('x:', x, '\nw:', w)
```

```
x: 500
w: 807
```

Importantly, `foo` "remembers" that it was created in the global environment, so looks in the global environment to find a value for `x` . It does **not** look back in its "call chain"; rather, it looks back in its parent environment.

Optional arguments and default values

In [24]:
```
def foo(x, y = []):
    y = y + [x]
    return y

a = foo(7)
b = foo(8, [1, 2, 3])
print('a:', a, '\nb:', b)
```

```
a: [7]
b: [1, 2, 3, 8]
```

In [25]:
```
c = foo(7)
print('a:', a, '\nb:', b, '\nc:', c)
```

```
a: [7]
b: [1, 2, 3, 8]
c: [7]
```

Let's try something that looks close to the same thing... but with an important difference!

In [26]:
```
def foo(x, y = []):
    y.append(x)   # different here
    return y

a = foo(7)
b = foo(8, [1, 2, 3])
print('a:', a, '\nb:', b)
```

```
a: [7]
b: [1, 2, 3, 8]
```

Okay, so far it looks the same as with the earlier `foo`.

In [27]:
```
c = foo(7)
print('a:', a, '\nb:', b, '\nc:', c)
```

```
a: [7, 7]
b: [1, 2, 3, 8]
c: [7, 7]
```

So quite different... all kinds of aliasing going on. Perhaps surprisingly, the default value to an optional argument is only evaluated once, at function *definition* time. The moral here is to be **very** careful (and indeed it may be best to simply avoid) having optional/default arguments that are mutable structures like lists... it's hard to remember or debug such aliasing!

Reference Counting

This is an advanced feature you don't need to know about, but you might be curious about. Python knows to throw away an object when its "reference counter" reaches zero. You can inspect the current value of an object's reference counter with `sys.getrefcount`.

In [28]:
```
import sys
L1 = [301, 302, 303]
print(sys.getrefcount(L1))
L2 = L1
print(sys.getrefcount(L1))
L3 = [L1, L1, L1]
print(sys.getrefcount(L1))
L3.pop()
print(sys.getrefcount(L1))
L3 = 307
print(sys.getrefcount(L1))
```

```
2
3
6
5
3
```

Readings -- if you want/need more refreshers

Check out readings and exercises from **6.145**:

- Assignment and aliasing
- What is an environment? What is a frame? How should we draw environment diagrams?
- What is a function? What happens when one is defined? What happens when one is called?
- What happens when a function is defined inside another function (also known as a closure)?
- What is a class? What is an instance? What is self? What is __init__?
- How does inheritance in classes work?

Another resource is the Think Python textbook.