notional_machine

February 3, 2020

1 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 (in particular, keeping track of variable bindings).

1.1 Variables and data types

1.1.1 Integers

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

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

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

1.1.2 Lists

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

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

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

Unlike integers, lists are mutable:

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

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

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

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

[ ]:
```
b[0][0] = 304
print(b)
```

1.1.3 Tuples

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

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

Unlike a list, we can't change the top most structure of a tuple; trying to change it results in an error:

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

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

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

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.

1.1.4 Strings

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

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

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

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

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.

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

[ ]:
```
x = [301, 302, 303]
y = x
x.append([304, 305])
print('x:', x, '\ny:', y)
```

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

[ ]:
```
x = [301, 302, 303]
y = x
x.extend([304, 305])
print('x:', x, '\ny:', y)
```

What happens when using the '+' operator used on lists?

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

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?

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

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.

1.2 Functions and scoping

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

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

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.

1.2.1 Optional arguments and default values

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

a = foo(307)
b = foo(388, [301, 302, 303])
print('a:', a, '\nb:', b)
```

[ ]:
```
c = foo(307)
print('a:', a, '\nb:', b, '\nc:', c)
```

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

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

a = foo(307)
b = foo(388, [301, 302, 303])
print('a:', a, '\nb:', b)
```

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

[ ]:
```
c = foo(307)
print('a:', a, '\nb:', b, '\nc:', c)
```

So quite different… all kinds of aliasing going on. 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!

1.3 Closures

[ ]:
```
def add_n(n):
    def inner(x):
        return x + n
    return inner
```

[ ]:
```
add1 = add_n(301)
add2 = add_n(302)

print(add2(303))
print(add1(307))
print(add_n(308)(309))
```