Lecture 03.1 DataTypes

September 1, 2022

1 Data Types

Duncan Callaway

This notebook is a review of variable types. My objective is to review enough of the basics to get you in a position to understand how a Pandas dataframe works, which is upcoming

1.1 Object, method, function?

These phrases come up repeatedly in python.

1.1.1 Q: What are they?

Go here for more

1. Object: virtually anything with attributes in python.
   1. Objects usually belong to classes and have attributes
   2. For example an object might belong to a bike class. The attributes of the class would be material, wheel size, number of gears, etc.
2. Method: A function associated with an object.
   a. for example `object.method` applies the method to the object.
3. Function: performs an action using some set of input parameters.
   b. for example `function(object)` applies the function to the object.

We'll talk about each of these in class today.

1.2 Basic data, or variable, types

1.2.1 Q: what are some variable types native to Python?

1. `string`
2. `numbers`
   1. `int`
   2. `float`
3. `bool`

Strings

```
[11]: name = 'Imogen'
      type(name)
```

```
[11]: str
```

```
[12]: name[2]
```

```
[12]: 'o'
```

1.2.2 Q: In class action: Change the e in Imogen to y.

What you'll find: though we can index their contents, strings are *immutable*, meaning you can't change individual elements. Instead you need to do a wholesale reassignment.

```
[13]: name[4] = 'y'
```

```
            ␣
 ↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-13-b6910fa71c71> in <module>
    ----> 1 name[4] = 'y'


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

```
[14]: name = name[0:4] + 'y' + name[5]
      name
```

```
[14]: 'Imogyn'
```

Floats

```
[20]: x = 0.34
      type(x)
```

```
[20]: float
```

Int

```
[21]: i = 1
      type(i)
```

```
[21]: int
```

Boolean

[29]:
```
ans = True
type(ans)
```

[29]:
```
bool
```

Note that Python is 'dynamically typed' meaning it assigns a variable a type based on what you set it equal to.

You don't need to define the type ahead of time.

Note this can cause problems, as you can mistakenly assign variables values that won't play nicely later on.

1.3 Data structures or "containers"

1.3.1 Q: what are some data structures native to Python?

1. `dict`
2. `list`
3. `tuple`
4. `set`

Let's explore these a little further.

1.4 Lists

[30]:
```
squares = [1, 5, 9, 16, 25] # do this in lecture
squares
```

[30]:
```
[1, 5, 9, 16, 25]
```

We can index elements with the usual process:

[31]:
```
print(squares[0]) # do this in lecture; zero is the first element
print(squares[-1]) # do this in lecture; -1 gives the last
print(squares[2:]) # do this in lecture; you can slice...
```

```
1
25
[9, 16, 25]
```

1.4.1 Q: What does "mutable" mean?

…that you can change individual entries of the data structure.

Lists are mutable:

[32]:
```
squares[1] = 4
squares
```

[32]:
```
[1, 4, 9, 16, 25]
```

That's better!

1.4.2 Q: In class, append 36 to your list.

We can also append:

```
[33]: squares.append(6**2)
      squares
```

```
[33]: [1, 4, 9, 16, 25, 36]
```

We can append lists this way too:

```
[34]: squares = squares + [49]
      squares
```

```
[34]: [1, 4, 9, 16, 25, 36, 49]
```

Finally, we can nest lists

```
[35]: x = [1,2,3]
      y = [4,6]
```

```
[36]: A = [x,y]
      A
```

```
[36]: [[1, 2, 3], [4, 6]]
```

Note that the number of elements in the two lists within the list did not need to be equal.

Note also that we can also assign different variable types to the nested list:

```
[37]: a = ['abc', 'def']
```

```
[38]: A = [x,a]
      A
```

```
[38]: [[1, 2, 3], ['abc', 'def']]
```

1.4.3 Q: Index the matrix A to get out 'def'

It's tough to index these things.

Think about how you might get the entry 'def' from the list, by integer indexes.

```
[39]: A[1,1]
```

```
        ␣
↪---------------------------------------------------------------------------
```

```
        TypeError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-39-80a1a5f00f0b> in <module>
    ----> 1 A[1,1]


        TypeError: list indices must be integers or slices, not tuple
```

That didn't work! (We have to wait for numpy and pandas to be able to do that.)

```
[40]: A[1][1]
```

```
[40]: 'def'
```

Note, pandas data frames are a lot like lists in this way. You first index which "sublist" you want, then you index into elements of that. Numpy arrays (as we'll see) are much easier to index. Frankly, the indexing with pandas is pretty annoying, but there are some workarounds that we'll discuss.

1.5 Tuples

Tuples are like lists – they hold multiple elements and you can index them.

```
[43]: B = (1,2,3)
      B
```

```
[43]: (1, 2, 3)
```

```
[44]: B[1]
```

```
[44]: 2
```

1.5.1 Q: Are tuples mutable?

A: nope!

```
[45]: B[1] = 3
```

```
      ␣
 ↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-45-9bc40c1a86bc> in <module>
    ----> 1 B[1] = 3
```

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

Changing and individual element throws an error.

So why bother with tuples? 1. They are harder to work with, but 2. They prevent you from doing things you don't want to, and 3. They are more memory-efficient.

1.6 Sets

Whereas lists are defined with square brackets, sets use curlies:

```
[46]: basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
```

What happened there?

...I wrote orange and apple twice. Let's look at the set:

```
[47]: print(basket)
```

```
{'banana', 'pear', 'apple', 'orange'}
```

Pretty smart – no duplicates!

1.6.1 Q: What might you do if you have a variable called basket but you don't know what's in it?

Now we can query the set for membership:

```
[49]: 'orange' in basket
```

```
[49]: True
```

```
[50]: 'kiwi' in basket
```

```
[50]: False
```

1.6.2 Q: Try indexing the list

```
[58]: basket[1]
```

```
            ␣
 ↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-58-b034c74c3c8f> in <module>
```

```
----> 1 basket[1]


      TypeError: 'set' object is not subscriptable
```

1.6.3 Q: Why can't you index a set?

Ans: because the *order* or elements in the set is ambiguous, especially when you think about how duplicates get handled.

1.7 Dict

Note, pandas data frames are like lists in that there are a number of entries that we can very quickly query. But there are even more similarities with `dict` data structures.

With dicts, we are inching closer to the kind of structure we have in a pandas data frame.

And we get a way to index elements of a set, by associating one value with another.

[59]:
```
cars = {'Prius':'Toyota', 'Volt':'Chevy', 'Model 3': 'Tesla'}
cars
```

[59]:
```
{'Prius': 'Toyota', 'Volt': 'Chevy', 'Model 3': 'Tesla'}
```

1.7.1 Q: after typing in your own dict, try indexing it to get information back out.

[60]:
```
cars['Prius']
```

[60]:
```
'Toyota'
```

In this case 1. 'Prius' is the **key** of the dict. 2. 'Toyota' is the associated **value**.

We can add new entries very easily:

[61]:
```
cars['Leaf'] = 'Nissan'
```

[62]:
```
cars
```

[62]:
```
{'Prius': 'Toyota', 'Volt': 'Chevy', 'Model 3': 'Tesla', 'Leaf': 'Nissan'}
```

Important note here: the dict does not store entries in any particular order – you rely on the key to pull data out.