6.00.1.x - Week 3 - Structured Types

September 24, 2020

1 Video 1: Tuples

- An ordered sequence of elements, can mix element types.
- **Immutable**, cannot change element values.
- Represented with parentheses.

Something strange happens when you try to invoke part of the tuple.

[2]:
```
t = (2,"one",3)
t[0]
```

[2]:
```
2
```

[3]:
```
t[1:2]
```

[3]:
```
('one',)
```

[4]:
```
t[1:3]
```

[4]:
```
('one', 3)
```

- They can be conveniently used to swap variables:

(x,y) = (y,x)

This works perfectly well and no information is lost.

- Tuples can be used to return more than one value from a function, using the command **return** (q,r)
- Tuples can be used to iterate over them.

When a tuple has only one element, you must specify it as follows: (elt,).

2 Video 2: Lists

- Ordered sequence of information, accessible by index.
- A list is denoted by square brackets.
- A list contains elements (usually homogeneous, can contained mixed types too).
- List elements can be changed so a list is **mutable**.

- The index can be a **Variable or expression**, must evaluate to an int.
- I can iterate over a list too.

3 Video 3: List Operations

- **Add** elements to end of list with L.append(element). This **mutates** the list!

[7]:
```
L = [2,1,3]
L.append(5)
print(L)
```

```
[2, 1, 3, 5]
```

What is the dot?

- Lists are Python objects, everything in Python is an object.
- Objects have data.
- Objects have methods and functions.
- Access this information by object_name.do_something()

- Can also use **concatenation** with the + operator.

[9]:
```
L1 = [2,1,3]
L2 = [4,5,6]
L3 = L1 + L2
print(L3)
```

```
[2, 1, 3, 4, 5, 6]
```

- Can also **mutate** with the command extend.

[10]:
```
L1.extend([0,6])
print(L1)
```

```
[2, 1, 3, 0, 6]
```

- Can also **delete an element** with del(L[index]).
- Remove element at **end of list** with L.pop(), returns the removed element.

[12]:
```
L = [2,1,3,6,3,7,0]
L.remove(2)
print(L)
L.remove(3)
print(L)
del(L[1])
print(L)
L.pop()
print(L)
```

```
[1, 3, 6, 3, 7, 0]
[1, 6, 3, 7, 0]
```

```
[1, 3, 7, 0]
[1, 3, 7]
```

- Convert **string to list** with list(s), returns a list with every character from s an element in L.
- Can use s.split(), to split a string on a character parameter, splits on spaces if called without a parameter.

Other operations are: sort(), sorted(), reverse(), and many more.

https://docs.python.org/2/tutorial/datastructures.html

[13]:
```
# Difference between sort and sorted
L = [9,6,0,3]
sorted(L)
```

[13]:
```
[0, 3, 6, 9]
```

[14]:
```
L.sort()
print(L)
```

```
[0, 3, 6, 9]
```

[15]:
```
L.reverse()
print(L)
```

```
[9, 6, 3, 0]
```

- **range** is a special procedure:
  - Returns something that behaves like a tuple.
  - Doesn't generate all elements at once, rather it generates the first element, and provides an iteration method by which subsequent elements can be generated.
  - While in a foor loop, what the loop variable iterates over behaves like a list.

4 Video 4: Mutation, Aliasing, Cloning

Lists in memory

- Mutable.
- Behave differently than immutable types.
- Is an object in memory.
- Variable name points to object.
- Any variable pointing to that object is affected.
- Key phrase to keep in mind when working with lists is **side effects**.

We are gonna look into three ideas: first aliasing, then mutation, then cloning.

[16]:
```
warm = ['red','yellow','orange']
hot = warm
print(warm)
print(hot)
```

```
['red', 'yellow', 'orange']
['red', 'yellow', 'orange']
```

[17]:
```
hot.append('pink')
print(hot)
print(warm)
```

```
['red', 'yellow', 'orange', 'pink']
['red', 'yellow', 'orange', 'pink']
```

hot is an **alias** for *warm.

Another concept: "If two lists print the same thing, that does not mean they are the same structure."

This is because there are 2 different slots in memory, one being accesed through each one of these handles.

[18]:
```
cool = ['blue','green','grey']
chill = ['blue','green','grey']
print(cool)
print(chill)
```

```
['blue', 'green', 'grey']
['blue', 'green', 'grey']
```

[19]:
```
cool.append('white')
print(cool)
```

```
print(chill)
```

```
['blue', 'green', 'grey', 'white']
['blue', 'green', 'grey']
```

Another idea in this realm is **cloning a list**, which creates a new list and **copies every element**.

[22]:
```
cool = ['blue','green','grey']
chill = cool[:]
print(cool)
print(chill)
```

```
['blue', 'green', 'grey']
['blue', 'green', 'grey']
```

[23]:
```
chill.append('black')
print(chill)
print(cool)
```

```
['blue', 'green', 'grey', 'black']
['blue', 'green', 'grey']
```

Sorting lists

- calling sort() **mutates** the list, returns nothing.
- calling sorted() **does not mutate** list, must assign result to a variable.

Careful with lists of lists!

[24]:
```
warm = ['yellow','orange']
hot = ['red']
brightcolors = [warm]
print(warm)
print(hot)
print(brightcolors)
```

```
['yellow', 'orange']
['red']
[['yellow', 'orange']]
```

Now let's add something to brightcolors

[25]:
```
brightcolors.append(hot)
print(brightcolors)
```

```
[['yellow', 'orange'], ['red']]
```

[26]:
```
hot.append('pink')
print(hot)
```

```
['red', 'pink']
```

[27]:
```
print(brightcolors)
```

```
[['yellow', 'orange'], ['red', 'pink']]
```

When I change hot, I also indirectly changed brightcolors. This is **aliasing** and we need to be careful about it.

[28]:
```
print(hot+warm)
```

```
['red', 'pink', 'yellow', 'orange']
```

Lastly, avoid mutating a list as you are iterating over it.

[29]:
```
def remove_dups(L1,L2):
    for e in L1:
        if e in L2:
            L1.remove(e)
L1 = [1,2,3,4]
L2 = [1,2,5,6]
remove_dups(L1,L2)
print(L1)
```

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

Why was number 2 not deleted?

- Python uses an internal counter to keep track of the index it is in the loop.
- Mutating changes the list length but Python doesn't update the counter.
- Loop never sees element 2.

How do I solve this? I make a copy before mutating it.

[30]:
```
def remove_dups_new(L1,L2):
    L1_copy = L1[:]
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
L1 = [1,2,3,4]
L2 = [1,2,5,6]
remove_dups_new(L1,L2)
print(L1)
```

```
[3, 4]
```

5 Video 5: Functions as Objects

Functions are **first class objects**:

- Have types.
- Can be elements of data structures like lists.
- Can appear in expressions: part of an assignment statement, or as an argument to a function.

Particularly useful to use functions as arguments when coupled with lists, aka **higher order programming**.

[3]:
```
def applyToEach(L, f):
    """assumes L is a list, f a function
    mutates L by replacing each element,
    e, of L by f(e)"""
    for i in range(len(L)):
        L[i] = f(L[i])

L = [1,-2,3.4]
applyToEach(L,abs)
print(L)
applyToEach(L,int)
print(L)
```

```
[1, 2, 3.4]
[1, 2, 3]
```

It could be done the other way too, applying a list of functions to a given argument. This idea can be generalized.

Python provides a general purpose HOP (higher order procedure), *map*.

[5]:
```
map(abs,[1,-2,3,-4])
# Produces an iterable
for elt in map(abs,[1,-2,3,-4]):
    print(elt)
```

```
1
2
3
4
```

Another example is the following:

[6]:
```
L1 = [1,28,36]
L2 = [2,57,9]
for elt in map(min,L1,L2):
    print(elt)
```

1
28
9

6 Video 6: Quick Review

Strings, tuples, ranges, lists

- Can get the ith element.
- Can get its length.
- Can concatenate them (not range).
- Can repeat them n times (not range).
- Can slice them.
- Can ask if an element is within them.
- Can iterate over its elements.
- Only lists are **mutable**.

7 Video 7: Dictionaries

Using multiple lists to keep information can be **messy**. It forces me to maintain many lists and use integers as indexes. It also forces me to remember to change multiple lists everytime I need to make a change.

Dictionaries

- Nice to index item of interest directly (not necesarily in *int*). We call the index *keys* when refering to dictionaries.
- Nice to use one data structure instead of many separate lists.
- Stores pairs of data - key & value.

[2]:
```
grades = {'Ana':'B', 'John':'A+', 'Denise':'A', 'Katy':'A'}
print(grades['John'])
```

```
A+
```

[3]:
```
grades['Sylvan'] = 'A'
print(grades['Sylvan'])
```

```
A
```

[4]:
```
# Can test if a key is in dictionary
'John' in grades
```

[4]: True

[5]:
```
'Jorge' in grades
```

[5]: False

[6]:
```
# Remove an entry
del(grades['Ana'])
print(grades)
```

```
{'John': 'A+', 'Denise': 'A', 'Katy': 'A', 'Sylvan': 'A'}
```

[7]:
```
# Can also get the keys - Gives me an iterable!
grades.keys()
```

[7]: dict_keys(['John', 'Denise', 'Katy', 'Sylvan'])

[ ]:
```
# Can also get the values - Gives me an iterable!
grades.values()
```

Values

- Can be any type (immutable and mutable).
- Can be duplicates.
- Can be lists, even other dictionaries.

Keys

- Must be **unique**.
- **Immutable** type (int, float, string, tuple, bool).
- Careful with float type as a key.
- **No order** to keys or values!

8 Video 8: Example with a Dictionary

1. Create a frequency dictionary.
2. Find word that occurs the most and how many times.
3. Find the words that occur at least X times.

[12]:
```
def lyrics_to_frequencies(lyrics):
    myDict = {}
    for word in lyrics:
        if word in myDict:
            myDict[word] += 1
        else:
            myDict[word] = 1
    return myDict


she_loves_you = ['she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',

'you', 'think', "you've", 'lost', 'your', 'love',
'well', 'i', 'saw', 'her', 'yesterday-yi-yay',
"it's", 'you', "she's", 'thinking', 'of',
'and', 'she', 'told', 'me', 'what', 'to', 'say-yi-yay',

'she', 'says', 'she', 'loves', 'you',
'and', 'you', 'know', 'that', "can't", 'be', 'bad',
'yes', 'she', 'loves', 'you',
'and', 'you', 'know', 'you', 'should', 'be', 'glad',

'she', 'said', 'you', 'hurt', 'her', 'so',
'she', 'almost', 'lost', 'her', 'mind',
'and', 'now', 'she', 'says', 'she', 'knows',
"you're", 'not', 'the', 'hurting', 'kind',

'she', 'says', 'she', 'loves', 'you',
'and', 'you', 'know', 'that', "can't", 'be', 'bad',
'yes', 'she', 'loves', 'you',
'and', 'you', 'know', 'you', 'should', 'be', 'glad',

'oo', 'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'with', 'a', 'love', 'like', 'that',
'you', 'know', 'you', 'should', 'be', 'glad',

'you', 'know', "it's", 'up', 'to', 'you',
'i', 'think', "it's", 'only', 'fair',
```

```
'pride', 'can', 'hurt', 'you', 'too',
'pologize', 'to', 'her',

'Because', 'she', 'loves', 'you',
'and', 'you', 'know', 'that', "can't", 'be', 'bad',
'Yes', 'she', 'loves', 'you',
'and', 'you', 'know', 'you', 'should', 'be', 'glad',

'oo', 'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'she', 'loves', 'you', 'yeah', 'yeah', 'yeah',
'with', 'a', 'love', 'like', 'that',
'you', 'know', 'you', 'should', 'be', 'glad',
'with', 'a', 'love', 'like', 'that',
'you', 'know', 'you', 'should', 'be', 'glad',
'with', 'a', 'love', 'like', 'that',
'you', 'know', 'you', 'should', 'be', 'glad',
'yeah', 'yeah', 'yeah',
'yeah', 'yeah', 'yeah', 'yeah'
]

beatles = lyrics_to_frequencies(she_loves_you)
print (beatles)
```

```
{'she': 20, 'loves': 13, 'you': 36, 'yeah': 28, 'think': 2, "you've": 1, 'lost':
2, 'your': 1, 'love': 5, 'well': 1, 'i': 2, 'saw': 1, 'her': 4, 'yesterday-yi-
yay': 1, "it's": 3, "she's": 1, 'thinking': 1, 'of': 1, 'and': 8, 'told': 1,
'me': 1, 'what': 1, 'to': 3, 'say-yi-yay': 1, 'says': 3, 'know': 11, 'that': 7,
"can't": 3, 'be': 10, 'bad': 3, 'yes': 2, 'should': 7, 'glad': 7, 'said': 1,
'hurt': 2, 'so': 1, 'almost': 1, 'mind': 1, 'now': 1, 'knows': 1, "you're": 1,
'not': 1, 'the': 1, 'hurting': 1, 'kind': 1, 'oo': 2, 'with': 4, 'a': 4, 'like':
4, 'up': 1, 'only': 1, 'fair': 1, 'pride': 1, 'can': 1, 'too': 1, 'pologize': 1,
'Because': 1, 'Yes': 1}
```

[15]:
```
# How to define the most common words?

def most_common_words(freqs):
    # First I find the number of repetition
    values = freqs.values()
    best = max(freqs.values())
    words = []
    # Now I find which words reached that number
    for k in freqs:
        if freqs[k] == best:
            words.append(k)
    return (words, best)

(w,b) = most_common_words(beatles)
```

```
print(w,b)
```

```
['her', 'with', 'a', 'like'] 4
```

[17]:
```
def words_often(freqs, minTimes):
    result = []
    done = False
    while not done:
        temp = most_common_words(freqs)
        if temp[1] >= minTimes:
            result.append(temp)
            for w in temp[0]:
                del(freqs[w])  #remove word from dictionary
        else:
            done = True
    return result

print(words_often(beatles, 5))
```

```
[]
```

9 Video 8: Fibonacci and Dictionaries

Going back to standard Fibonacci code to see why it is **inefficient**.

[ ]:
```
def fib(n):
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return fib(n-1) + fib(n-2)
```

It is inefficient because many calculations are repeated over and over unnecesarily. It really starts getting slower.

Instead of recalculating the same values many times, we can keep track of what we've already done.

[18]:
```
def fib_efficient(n, d):
    # Do a lookup first in case I've already calculated the value
    if n in d:
        return d[n]
    else:
        # Modify dictionary as progress through function calls
        ans = fib_efficient(n-1, d) + fib_efficient(n-2, d)
        d[n] = ans
        return ans

# The base cases need to be added to the initial dictionary
d = {1:1, 2:2}
print(fib_efficient(6, d))
```

```
13
```

This is often called **memoization**: using the dictionary to hold on to values that I've already computed and I don't need to compute again.

10 Video 9: Global Variables

Can be dangerous to use:

- Breaks the scoping of variables by function call.
- Allows for side effects of changing variables values in ways that affect other computation.

But can be convenient when want to keep track of information inside a function. Let's use the Fibonacci efficient and standard form of the function to test this concept.

[27]:
```
# Standard way
def fib(n):
    global numFibCalls
    numFibCalls += 1
    if n == 1:
        return 1
    elif n == 2:
        return 2
    else:
        return fib(n-1)+fib(n-2)

# Efficient way
def fibef(n, d):
    global numFibCalls
    numFibCalls += 1
    if n in d:
        return d[n]
    else:
        ans = fibef(n-1,d)+fibef(n-2,d)
        d[n] = ans
        return ans

# Testing the idea
numFibCalls = 0
fibArg = 30

print(fib(fibArg))
print('function calls', numFibCalls)

numFibCalls = 0

d = {1:1, 2:2}
print(fibef(fibArg, d))
print('function calls', numFibCalls)
```

```
1346269
function calls 1664079
1346269
function calls 57
```