Middlebury

CSCI 146: Intensive Introduction to Computing

Fall 2025

Lecture 11: Tuples and `Python`'s Memory Model

Goals for today

- Create and use a tuple.
- Use tuples to return multiple values from a function.
- Describe the Python memory model.
- Explain the behavior of references.
- Predict how immutable and mutable values referenced by variables will be affected by assignments, operators, functions and methods.

One operator I forgot to mention about sets: < (strict/proper subset)

```
>>> set("ab") < set("cba")
True
>>> set("ab") < set("ab")
False
>>> set("ab") <= set("ab")
True
```

Warmup: write a program to count the frequency of words in `"here_comes_the_sun.txt"` linked on the website.

Work in pairs?

- In `VS Code`: `View -> Extensions`, search for `LiveShare` (by Microsoft) and **Install**.
- ONE person: click on on the `LiveShare` button at the bottom.
- The sharing link is copied to the clipboard: email/text to your partner!

```
frequency = {} # initialize dictionary
with open("here_comes_the_sun.txt", "r") as file:
    for line in file:
            line = line.strip()
            if line == "":
                continue
            words = line.split()
            for word in words:
              word = word.lower()
              if word in frequency:
                frequency[word] += 1
              else:
                frequency[word] = 1
```

Introducing `tuple`s: an *immutable* sequence that can hold any type.

Why???

- Can't increase in size, so a `tuple` can represent a fixed collection of data with an expected structure, e.g. `("October", 15, 2025)`.
- Delimited with parentheses `()`.

```
>>> my_tuple = (1, 2, 3, 4)
>>> my_tuple
(1, 2, 3, 4)
>>> another_tuple = ("a", "b", "c", "d")
>>> another_tuple
('a', 'b', 'c', 'd')
>>> my_tuple[0]
1
>>> my_tuple[1]
2
>>> for i in my_tuple:
...     print(i)
1
2
3
4
>>> my_tuple[1:3]
(2, 3)
```

We can also "unpack" which is useful for swapping stuff.

Unpacking:

```
>>> var1, var2, var3, var4 = another_tuple
>>> var1
'a'
>>> var4
'd'
```

Swapping values:

```
>>> x = 10
>>> y = 20
>>> (y, x) = (x, y)
>>> x
20
>>> y
10
```

But `tuple`s are immutable:

```
>>> my_tuple[0] = 5 # TypeError: 'tuple' object does not support item assignment
>>> my_tuple.append(5) # AttributeError: 'tuple' object has no attribute 'append'
```

Back to our `frequency` analysis: how can we list the most frequent words in descending order?

```
# build up a list of word pairs from the dictionary we created
word_pairs = []
for word, count in frequency.items():
    word_pairs.append((count, word)) # note the use of ()

# sort in descending order
# Python will compare pairs lexicographically (i.e. count first, then word)
word_pairs.sort(reverse=True)
for count, word in word_pairs:
    print(word + " appears " + str(count) + " times")
```

Be careful when trying to modify a container (e.g. `dict`) while iterating.

This is NOT okay!

```
d = { 1 : "one", 2 : "two", 3 : "three" }
for key in d:
    if key == 2:
        d.pop(key) # RuntimeError: dictionary changed size during iteration
```

But this *is* okay...

```
d = { 1 : "one", 2 : "two", 3 : "three" }
for key in [1, 2, 3]:
    if key == 2:
        d.pop(key)
```

Why??

- In the first example, the iteration variable depends on the dictionary and it becomes invalidated when the dictionary is modified.
- In the second example, the iteration sequence is a snapshot of the keys and decoupled from the dictionary.

In any case, it would be better to work with a *copy* of a dictionary if we need to modify it when looping.

...but we also need to be careful with copying...

- Almost everything in `Python` is an object (instance of a class, i.e. a house made from a blueprint).
- Objects take up memory in our computers.
- `Python` keeps track of the *references* to these objects (like the address of a house built from a blueprint).

Tracking references of our objects: immutable objects

```
>>> z = 2 # re-assignment creates a new object
>>> x = 1
>>> y = x # now x and y both "point" to the same object
>>> x = z
>>> x = "hello"
>>> y = x
>>> y = "bye"
>>> x
"hello"
>>> y
"bye"
```

Investigate the `id()` function!

Tracking references of our objects: mutable objects

```
>>> a = [1, 2, 3]
>>> b = a
>>> a[1] = 4
>>> a
[1, 4, 3]
>>> b
[1, 4, 3]
>>> b = [6, 7, 8] # creates a new list object
>>> a
[1, 2, 3]
>>> b
[6, 7, 8]
```

Investigate the `id()` function!

Parameters as references

```
def aliasing(param):
    param[1] = 4
a = [1, 2, 3]
aliasing(a)
a # a will be [1, 4, 3]

def my_function(a):
    a = [0]*5
    a[0] = 6
x = [1, 2, 3, 4, 5]
my_function(x)
x # x will be [1, 2, 3, 4, 5]
```

Shallow versus deep copying.

- Slicing creates a shallow copy: mutable items in the list are not deep-copied.
- If we really want a "deep" copy, we would need to use the `copy` module.

```
>>> x = [1, 2, 3, 4, 5]
>>> y = x[0:2]
>>> y
[1, 2]
>>> y[0] = 6
>>> y
[6, 2]
>>> x
[1, 2, 3, 4, 5]
>>> y = x[:]
>>> y[4] = 12
>>> y
[1, 2, 3, 4, 12]
>>> x
[1, 2, 3, 4, 5]
```

```
>>> x = [1, 2, [3, 4], 5, 6]
>>> y = x[:]
>>> y[2][0] = 7
>>> y[3] = 8
>>> y
[1, 2, [7, 4], 8, 6]
>>> x
[1, 2, [7, 4], 5, 6]
```

Question 1: After the code below executes, what is the value of `a`?

```
a = [[1, 2, 3], [4, 5]]
b = a[:]
b.append(8)
```

A. `[[1, 2, 3], [4, 5]]`

B. `[[1, 2, 3], [4, 5], 8]`

C. `[[1, 2, 3], [4, 5, 8]]`

D. `[[1, 2, 3], [4, 5], [8]]`

Question 2: After the code below executes, what is the value of `a`?

```
a = [[1, 2, 3], [4, 5]]
b = a[:]
b[1].append(8)
```

A. `[[1, 2, 3], [4, 5]]`

B. `[[1, 2, 3], [4, 5], 8]`

C. `[[1, 2, 3], [4, 5, 8]]`

D. `[[1, 2, 3], [4, 5], [8]]`

Exercise: here is a function that inserts a value `b` after every occurence of `a` in a list called `x`.

```
def insert_after(x, a, b):
    """
    Return a new list consisting of all elements from x,
    plus a copy of b after each occurrence of a.

    (list of int, int, int) -> list of int

    >>> insert_after([3, 4, 5], 3, 10)
    [3, 10, 4, 5]
    """
    new_x = []
    for element in x:
        new_x.append(element)
        if element == a:
            new_x.append(b)
    return new_x
```

Rewrite the function to insert the values in-place (it should not return a list anymore).

Possible solution.

```
def insert_after2(x, a, b):
    """
    Insert b after each occurrence of a in x.

    (list of int, int, int) -> NoneType

    >>> x = [3, 4, 5]
    >>> insert_after2(x, 3, 10)
    >>> x
    [3, 10, 4, 5]
    """
    i=0
    while i < len(x):
        if x[i] == a:
            x.insert(i + 1, b)
            i += 1
        i += 1
```

Summary and Reminders

- BE CAREFUL WITH MUTABLE OBJECTS (1) assigned to variables or (2) passed to functions.
- Programming Assignment 5 initially due tomorrow (can work and submit in pairs).
- Programming Assignment 3 final due date tomorrow.
- Use "Regrade Requests" form on the website. See Gradescope comments by clicking on **Code**.