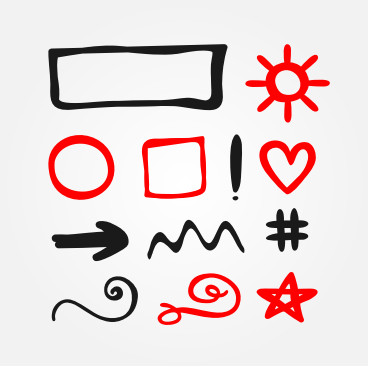

CPSC 5061—Programming I for Business

Worksheet 10

Tuples

Motivation

Strings were so successful in Python—a collection datatype that supports:

- indexing,
- slicing,
- for-loop iteration,
- membership with the `in` operator,
- knowing its length with the `len` built-in,
- literal constructions using quotation marks, and
- concatenation with the **+** and ***** operators.

Powerful!

Perhaps we could leverage that success by generalizing the string concept to a datatype that can contain not just letters as elements. That's just what the Python designers did.

Concept

The generalization of the `str` datatype that can hold elements of *any datatype* is a `tuple`. Tuples support the same set of operations that a string does listed above, but in a tuple, each element can be an object of arbitrary datatype.

Tuple Literals

Like strings, tuples can be constructed as literals in your program. As we know, strings use quotation marks to construct literals. Tuples instead use a comma-separated list of expressions enclosed in parentheses.

```
s = 'abc'              # literal string with three elements
t = (1, 3.8, 'girl')   # literal tuple with three elements
```

Unlike a string where the elements are letters, a tuple can have any object and so in a tuple we can use any Python expressions to calculate each element's value.

```
n = 2
t = (n, n * n, n**3)  # the tuple (2, 4, 8)
```

Python evaluates these expressions and then constructs the tuple. This is similar to how it calculates arguments before calling a function.

Just like we can have an empty string with no letters, so, too, we can have an empty tuple with no elements. The literal for this is just an empty pair of parentheses.

```
t = ()  # t is now the empty tuple
```

Because Python uses parentheses for grouping, too, it would have an ambiguity between a one-element tuple and a grouped expression. To resolve this ambiguity, it makes construction of a one-element tuple require a *nonsensical comma* immediately before the closing parenthesis.

```
t = (68,)  # t is a now a tuple with a single element, 68
```

Contrast that to the following use of parentheses which Python interprets as a mere indication of grouping.

```
n = (68)  # equivalent to n = 68; n is an integer, not a tuple
```

Practice

1. Assign a tuple to **t** which has three elements: 0, -6.32 x 10^{7}, and -17.

2. Assign a tuple to **x** which has four elements: the current value of **n**, the square root of **n**, and the strings `'foo'` and `'bar'` concatenated with the string in **`suffix`**.

3. Assign the empty tuple to **z**.

4. Assign the tuple with the sole element -3.8 x 10^{-21} to the variable **one**.

5. Assign a tuple to **w** where the first element is the number **6**, the second element is the tuple **`('a', -3, 4)`**, and the last element is **`range(100)`**.

6. Assign a tuple to **v** which is a triple whose elements are each the number **1**.

7. Assign a tuple to **y** a one-element tuple whose element is the value of **`3 * n`**.

8. Assign a tuple to **f** whose elements are **x**, **y**, and **z**.

Indexing, Slicing, and Length

Like strings, ranges, and most other collection datatypes, tuples can be indexed and sliced using the square bracket notation you are already familiar with. They can also reveal their element count with the built-in `len`.

```
>>> t = (1, 99, -4, 16, 491)
>>> t[0]
1
>>> t[-1]
491
>>> t[:2]
(1, 99)
>>> t[2::2]
(-4, 491)
>>> len(t)
5
>>> (len(t), len(t[2::2]), len(t[:2]), len(()), len((1,)))
(5, 2, 2, 0, 1)
```

All these operations can be combined in any way. Python just does one thing after the other until the expression is fully evaluated.

```
>>> (1, 'abc', 4.2)[-1]
4.2
>>> ((1, 'abc', 4.2), (1, 3), (4, 5, 6, 7))[1]
(1, 3)
>>> ((1, 'abc', 4.2), (1, 3), (4, 5, 6, 7))[0][-2]
'abc'
```

Exercise

What are the results of the following expressions?

```
((1, 'xyzw', 4.2), (1, 3), (4, 5, 6, 7))[:2]
```

```
((1, 'xyzw', 4.2), (1, 3), (4, 5, 6, 7))[:2][0]
```

```
((1, 'xyzw', 4.2), (1, 3), (4, 5, 6, 7))[:2][0][-2]
```

```
((1, 'xyzw', 4.2), (1, 3), (4, 5, 6, 7))[:2][0][-2][1]
```

Concatenation

Again, follow what happens in strings to understand tuples.

```
s = 'abc' + 'xyzw'   # s is the new string 'abcxyzw'
```

Similarly,

```
t = (1, 2) + (4, 5, 6, -3)  # t is the new tuple (1, 2, 4, 5, 6, -3)
```

And, as with strings and in math, the ***** operator repeats the behavior of the **+** operator. So

```
t = (1,) * 4  # t is the new tuple (1, 1, 1, 1)
v = (1, 'a') * 3  # v is the new tuple (1, 'a', 1, 'a', 1, 'a')
```

Looping

Because of tuples' concatenation behavior like strings, we can use the pattern in the following example for building up ever more lengthy tuples:

```
print("What are your children's names?")
children = ()  ### start with an empty tuple
next_child = input()
while next_child != "":
    children += (next_child,)  ### build new tuple via concatenation
    next_child = input()
```

Note the two commented lines above. The first creates an empty tuple and assigns it to the variable **`children`**. Then within the loop, we construct a singleton tuple with each new child name, concatenate that with the previous tuple in **`children`**, and then replace **`children`** with this longer tuple.

When we're done, **`children`** will be a tuple of all the children's names. We might print them out one at a time with a for-loop. Note that a for-loop on a tuple behaves as you'd expect—it assigns the loop variable to each element in turn.

```
print("Your", len(children), "children's names are:")
for child in children:
    print(child)
```

Exercise

Practice the above looping patterns by inputting a series of numbers from the user until she types **`'DONE'`** and once you have them stored in a tuple of integers, print them out one at a time and also the cumulative sum at each step.