Tuples

- Tuples are essentially immutable lists
- Tuples are denoted with parentheses: `tup = (1,2,3)`
- Tuples store data in order
- Items in tuples are accessed via indexing and slicing
- Tuple items may not be changed. (However, if a tuple contains a modifiable object such as a list, the contained object may be modified)

Tuple examples

Tuples are seemingly similar to lists:

```
my_tuple = (1, 2, 3, 'str', 42.42)
print(my_tuple[1])
print(my_tuple[1:3])
```

We can loop over them with `for`:

```
for item in my_tuple:
    print(item)
```

Tuples are immutable

Unlike lists, we cannot change elements of a tuple.

```
my_list = ["I", "am", "a", "list"]
my_list[0] = "was"
print("my_list:", my_list)
```

```
my_tuple = ("I", "am", "a", "list")
my_tuple[3] = "tuple"
```

However...

Modifiable objects contained in a tuple are still modifiable:

```
my_tup = (2, 'a string', [1,3,8])
```

Here, `my_tup[2]` is a list, which is a mutable object.

We cannot reassign to the tuple:

```
my_tup[2] = 'something else'
```

But, we can modify a mutable object contained in a tuple:

```
my_tup[2][0] = 'new data'
print(my_tup)
```

Check out the behavior in Python Tutor.

Like all variables in Python, items in a tuple are actually references to objects in memory. Once a tuple has been created, its references to objects cannot be reassigned. Tuple items may reference an object that is mutable (say, a list). In this case the referenced mutable object may be changed in someway.

Exercise

Execute and understand the following code in Python Tutor.

```
sub_list_1 = [1,3,8]
sub_list_2 = ['z','y','x']
my_list = [2, 'a string', sub_list_1]
my_list[2] = sub_list_2
my_list[2][0] = 'new string'

my_tup = (2, 'a string', sub_list_1)

# cannot modify tuple references
# my_tup[2] = sub_list_2

# we can modify a mutable object referenced by a tuple
my_tup[2][0] = 'from my_tup'

# can can look at object ids
print(id(sub_list_1))
print(id(my_tup[2]))
```

A note on (im)mutability

It is natural to wonder why have immutable objects at all. One answer to this is practical: in Python, only immutable objects are allowed as keys in a dictionary or items in a set.

The more detailed answer requires knowledge of the underlying data structure behind Python dictionary and set objects. In the context of a Python dictionary, the memory location for a key-value pair is computed from a *hash* of the key. If the key were modified, the *hash* would change, likely requiring a move of the data in memory. Thus, Python requires immutable keys in dictionaries to prevent unnecessary movement of data.

It is possible to associate a value with a new key with the following code:

```
ages = {'cameron': 44, 'angelina': 40, 'bruce': 60, 'brad': 51, 'leo': 40}
print(ages)
ages['BRUCIE'] = ages['bruce']
del ages['bruce']
print(ages)
```

Tuple packing and unpacking

Tuple packing and unpacking is very convenient Python syntax. In packing, a tuple is automatically created by combining values or variables with commas:

```
t = 1, 2, 3
print("type(t):", type(t))
print("t:", t)
```

Tuple unpacking can store the elements of a tuple into multiple variables in one line of code.

```
my_tuple = ("a string", 43, 99.9)
my_str, my_int, my_flt = my_tuple
print(my_str)
print(my_int)
print(my_flt)
```

This is equivalent to:

```
my_tuple = ("a string", 43, 99.9)
my_str = my_tuple[0]
my_int = my_tuple[1]
my_flt = my_tuple[2]
```

Swapping variables via tuple packing

Tuple packing and unpacking allows swapping of variables without creating a temporary variable:

```
x = 1001
y = 'random string'
x, y = y, x
print("x:", x)
print("y:", y)
```