Tuples

- In Python tuples are very similar to lists, however, unlike lists they are *immutable* meaning they can not be changed.
- You would use tuples to present things that shouldn't be changed, such as days of the week, or dates on a calendar.
- You'll have an intuition of how to use tuples based on what you've learned about lists. We can treat them very similarly with the major distinction being that tuples are immutable.

Constructing Tuples

- The construction of a tuples use `()` with elements separated by commas.

```
# Create a tuple
my_tuple = (1,'a',3)
```

```
print(my_tuple)
```

```
(1, 'a', 3)
```

```
type(my_tuple)
```

```
tuple
```

```
# Can also mix object types
another_tuple = ('one',2, 4.53, 'asbc')

# Show
another_tuple
```

```
('one', 2, 4.53, 'asbc')
```

```
my_list = [1]
```

```
type(my_list)
```

```
list
```

```
my_tuple = (1,)
```

```
type(my_tuple)
```

```
tuple
```

```
my_tuple = 1,2,3
```

```
type(my_tuple)
```

```
tuple
```

```
my_tuple = (1,2,3,4)
```

```
len(my_tuple)
```

```
4
```

Tuple Indexing

- Indexing work just like in lists.
- A tuple index refers to the location of an element in a tuple.
- Remember the indexing begins from 0 in Python.
- The first element is assigned an index 0, the second element is assigned an index of 1 and so on and so forth.

```
print(another_tuple)
```

```
('one', 2, 4.53, 'asbc')
```

```
# Grab the element at index 0, which is the FIRST element
another_tuple[0]
```

```
# Grab the element at index 3, which is the FOURTH element
another_tuple[3]
```

```
# Grab the element at the index -1, which is the LAST element
another_tuple[-1]
```

```
# Grab the element at the index -3, which is the THIRD LAST element
another_tuple[-3]
```

```
2
```

Tuple Slicing

- We can use a `:` to perform *slicing* which grabs everything up to a designated point.
- The starting index is specified on the left of the `:` and the ending index is specified on the right of the `:`.
- Remember the element located at the right index is not included.

[ ] ↳ 15 cells hidden

Tuple Methods

- Tuples have built-in methods, but not as many as lists do.

[ ] ↳ 9 cells hidden

Immutability

- It can't be stressed enough that tuples are immutable.

```
print(my_tuple)
```

```
(1, 2, 3, 4, 5, 6, 1, 1, 2)
```

```
my_tuple[0]
```

```
1
```

```
my_tuple[0] = 'change'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-1113c19e61ee> in <module>()
----> 1 my_tuple[0] = 'change'

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

- Because of this immutability, tuples can't grow. Once a tuple is made we can not add to it.

- Let us consider a list and lets see if we can do this operation on them

```
# Create a list
my_list = [5,7,9,3,2]
```

```
# Replace the FIRST element with the value 1
my_list[0] = 1
```

```
# Our modified list
my_list
```

```
[1, 7, 9, 3, 2]
```

- Tuple does not support methods such as `append()`, `extend()`, `remove()`, `pop()`

```
my_tuple.append('Great!')
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-52-762f4e61335a> in <module>()
----> 1 my_tuple.append('Great!')

AttributeError: 'tuple' object has no attribute 'append'
```

`zip()`

- `zip()` function takes multiple lists as arguments and zips them together
- This function returns a list of n-paired tuples where n is the number of lists being zipped

```
city_list = ['Delhi','Patna','Cuttack','Guwahati']
river_list = ['Yamuna','Ganga','Mahanadi','Brahmaputra']
```

```
zip(city_list, river_list)
```

```
<zip at 0x7f8ef0d3e3c0>
```

```
city_and_their_rivers = list(zip(city_list,river_list))
```

```
city_and_their_rivers
```

```
[('Delhi', 'Yamuna'),
 ('Patna', 'Ganga'),
 ('Cuttack', 'Mahanadi'),
 ('Guwahati', 'Brahmaputra')]
```

```
city_list = ['Delhi','Patna','Cuttack','Guwahati']
river_list = ['Yamuna','Ganga','Mahanadi','Brahamputra','Thames']
```

```
list(zip(city_list,river_list))
```

```
[('Delhi', 'Yamuna'),
 ('Patna', 'Ganga'),
 ('Cuttack', 'Mahanadi'),
 ('Guwahati', 'Brahamputra')]
```

When to use Tuples

- You may be wondering, "Why bother using tuples when they have fewer available methods?" To be honest, tuples are not used as often as lists in programming, but are used when immutability is necessary. If in your program you are passing around an object and need to make sure it does not get changed, then a tuple becomes your solution. It provides a convenient source of data integrity.
- You will find them often in functions when you are returning some values

- You should now be able to create and use tuples in your programming as well as have an understanding of their immutability.

Sets

- Sets are an unordered collection of *unique* elements. We can construct them by using the `set()` function.
- Sets cannot have duplicates.
- Sets are mutable just like lists.
- You can create a non-empty set with curly braces by specifying elements separated by a comma.

```
# Create an empty set
empty_set = set()
```

```
type(empty_set)
```

```
set
```

```
# Create a non-empty set within curly braces
non_empty_set = {1,6,4,'abc'}
```

```
type(non_empty_set)
```

```
set
```

A Time for Caution

- An empty set cannot be represented as `{}`, which is reserved for an empty dictionary which we will get to know in a short while

```
my_object = {}
```

```
type(my_object)
```

```
dict
```

```
my_set = set()
```

```
type(my_set)
```

```
set
```

- We can cast a list with multiple repeat elements to a set to get the unique elements.

```
# Create a list with repeats
my_list = [1,1,2,2,3,4,5,6,1,1]
```

```
# Cast as set to get unique values
my_set = set(my_list)
```

```
my_set
```

```
{1, 2, 3, 4, 5, 6}
```

```
# A set cannot have mutable items
my_set = {1, 2, (3, 4)}
```

- We cannot create a set whose any of the elements is a list

```
# But we can have tuples as set elements, they are immutable
my_set = {1, 2, (2,3)}
```

```
my_set
```

```
{(2, 3), 1, 2}
```

```
my_set = {1,2,[3,5]}
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-653786077> in <cell line: 0>()
----> 1 my_set = {1,2,[3,5]}

TypeError: unhashable type: 'list'
```

Next steps: Explain error

```
my_set
```

```
{1, 2, 4}
```

`add()`

- `add()` method adds an element to a set
- This method takes the element to be added as an argument

```
# We add to sets with the add() method
my_set = set()
my_set.add('a')
```

```
#Show
my_set
```

```
{'a'}
```

```
# Add a different element
my_set.add(2)
```

```
#Show
print(my_set)
```

```
{2, 'a'}
```

```
# Lets add another element 1
my_set.add(1)
```

```
#Show
my_set
```

```
{1, 2, 'a'}
```

```
# Try to add the same element 1 again
my_set.add(1)
```

```
my_set
```

```
{1, 2, 'a'}
```

- Notice how it won't place another 1 there. That's because a set is only concerned with unique elements!

`update()`

- `update()` method helps to add multiple elements to a set

```
my_set = {5,7,9,3}
```

```
# add multiple elements
my_set.update([2, 3, 4])
print(my_set)
```

```
{2, 3, 4, 5, 7, 9}
```

`remove()`

- Use `remove()` to remove an item/element from the set.
- By default `remove()` removes the specified element from the set.
- `remove()` takes the element as an argument.

```
non_empty_set = {1,5,6,73,2}
```

```
non_empty_set.remove(5)
```

```
non_empty_set
```

```
{1, 2, 6, 73}
```

```
non_empty_set.remove(45)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-40-261421333> in <cell line: 0>()
----> 1 non_empty_set.remove(45)

KeyError: 45
```

Next steps: Explain error

- `remove()` throws an error when we try to remove an element which is not present in the set

`union()`

- `union()` method returns the union of two sets
- Also denoted by the operator `|`

```
# Initialize sets A and B
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
```

```
A.union(B)
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

```
A
```

```
{1, 2, 3, 4, 5}
```

```
# Also denoted by the operator |
A | B
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

`intersection()`

- `intersection()` method returns the intersection of two sets
- Also denoted by the operator `&`

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
```

```
A.intersection(B)
```

```
{4, 5}
```

```
# Also denoted by the operator &
A & B
```

```
{4, 5}
```

`difference()`

- `difference()` method returns the difference of two sets
- Difference of the set `B` from set `A` i.e, `(A - B)` is a set of elements that are only in `A` but not in `B`
- Also denoted by the operator `-`

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
```

```
A.difference(B)
```

```
{1, 2, 3}
```

```
# Also denoted by the operator -
A - B
```

```
{1, 2, 3}
```

```
B.difference(A)
```

```
{6, 7, 8}
```

```
B - A
```

```
{6, 7, 8}
```

`symmetric_difference()`

- `symmetric_difference()` method returns the set of elements in A and B but not in both (excluding the intersection)
- Also denoted by the operator `^`

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
```

```
A.symmetric_difference(B)
```

```
{1, 2, 3, 6, 7, 8}
```

```
A^B
```

```
{1, 2, 3, 6, 7, 8}
```

Dictionaries

- We've been learning about *sequences* in Python but now we're going to switch gears and learn about *mappings* in Python.
- If you're familiar with other languages you can think of these Dictionaries as hash tables.
- So what are mappings? Mappings are a collection of objects that are stored by a *key*, unlike a sequence that stored objects by their relative position. This is an important distinction, since mappings won't retain order since they have objects defined by a key.

- A Python dictionary consists of a key and then an associated value. That value can be almost any Python object. So a dictionary object always has elements as key-value pairs

Constructing a Dictionary

- A dictionary object is constructed using curly braces `{key1:value1,key2:value2,key3:value3}`

```
# Make a dictionary with {} and : to signify a key and a value
marvel_dict = {'Name':'Thor','Place':'Asgard','Weapon' : 'Hammer', 1:2, 3 : 'power', 'alibies' : ['Ironman','Captain America'], 'abc' : {1:2
```

```
# Call values by their key
marvel_dict['Place']
```

```
type(marvel_dict)
```

```
dict
```

```
marvel_dict['Name']
```

```
marvel_dict['Random']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-114-9f68b85bbfcd> in <module>()
----> 1 marvel_dict['Random']

KeyError: 'Random'
```

```
marvel_dict['Weapon']
```

```
marvel_dict['alibies']
```

```
['Ironman', 'Captain America']
```

```
type(marvel_dict['abc'])
```

```
dict
```

Dictionary Methods

[ ] ↳ 56 cells hidden