Tuples

- Commonly used in return functions
- Similar to lists
  - `tup = (1, 4, "hello")`
  - `len(tup)`
  - `tup[1]`
  - `for x in tup:`
- Immutable
- To modify a tuple
  - `tup = (1, 2, 3)`
  - `list_tuple = list(tup) # convert to list`
  - `list_tuple[0] = 9 # treat like a normal list`
  - `tuple(list_tuble) # convert back to a tuple`
- Unpacking tuples / lists
  - `a, b, c = tup # assign each ordered value from tuple / list to ordered variable`

Built-in Tuple Methods

Method	**Description**
`count()`	Returns the number of times a specified value occurs in a tuple
`index()`	Searches the tuple for a specified value and returns the position of where it was found

Sets

- `s = {1, 2, 3}`
- Unordered
- Unindexed
- Can't have duplicates
- Access item in set by looping
  - `for x in s:`
- Can't modify items already in the set
- `s.add(6) # adds 6 to set, if 6 is not already in it`
- `s.update(set_1) # add all items from set_1 to s`

  - `# set.update() works with any iterable object (tuples, lists, dictionaries, ...)`

- `s.discard(2) # removes the item from the set`

Built-in Set Methods

Method	Description
`add()`	Adds an element to the set
`clear()`	Removes all the elements from the set
`copy()`	Returns a copy of the set
`difference()`	Returns a set containing the difference between two or more sets
`difference_update()`	Removes the items in this set that are also included in another, specified set
`discard()`	Remove the specified item
`intersection()`	Returns a set, that is the intersection of two other sets
`intersection_update()`	Removes the items in this set that are not present in other, specified set(s)
`isdisjoint()`	Returns whether two sets have an intersection or not
`issubset()`	Returns whether another set contains this set or not
`issuperset()`	Returns whether another set contains this set or not
`pop()`	Removes an element from the set
`remove()`	Removes the specified element
`symmetric_difference()`	Returns a set with the symmetric differences of two sets
`symmetric_difference_update()`	Inserts the symmetric differences from this set and another
`union()`	Return a set containing the union of sets
`update()`	Update the set with the union of this set and others

Dictionaries

- Collection that uses key:value relationships
- `d = {"a":5, "b":8}`
- Ordered as of Python 3.7
- Mutable
- Can't have duplicate keys
- Can have duplicate values
- `d["a"] -> 5 # returns value associated with key "a"`

- `d["c"] = 10 # adds a new key:value pair`
- `len(d) -> 2 # returns the number of keys`
- `d["a"] = 4 # updates the value that the key "a" points to`
- `d.update({"a":4}) # updates the value the key "a" points to`
- `d.pop("a") # removes the key "a" and its relationship`
- `del d["a"] # removes the key "a" and its relationship`
- `for x in d: # loops through keys`
- `d.keys() -> all keys`
- `d.items() -> keys and items as tuples`
- `d.get("a", 2) -> returns the value for "a" if it exists, otherwise returns 2`

Dictionaries can be nested as well.

```
my_family = {
    "child_0": {
        "name": "Emily",
        "age": 8,
        "favorites": {
            "pet": "cat",
            "food": "cereal"
        },
    },
    "child_1": {
        "name": "Fred",
        "age": 14,
        "favorites": {
            "pet": "bird",
            "food": "pizza"
        },
    }
}
```

Dictionary Methods

Method	Description
`clear()`	Removes all the elements from the dictionary
`copy()`	Returns a copy of the dictionary
`fromkeys()`	Returns a dictionary with the specified keys and values
`get()`	Returns the value of the specified key
`items()`	Returns a list containing a tuple for each key value pair
`keys()`	Returns a list containing the dictionary's keys
`pop()`	Removes the element with the specified key
`popitem()`	Removes the last inserted key-value pair
`setdefault()`	Returns the value of the specified key. If the key does not exist: insert the key, with the value
`update()`	Updates the dictionary with the specified key-value pairs
`values()`	Returns a list of all the values in the dictionary

Dictionary Activity

Using a python script and command line to run

- Create a dictionary named `costs = {}`
- Add the key:value pair for the cost of a hat
  - `costs["hat"] = 20`
- Print the entire dictionary
  - `print(costs)`
- Add the costs of gloves and glasses
  - `costs["gloves"] = 15`
  - `costs["glasses"] = 30`
- Print the entire dictionary
- Print the cost of gloves
  - `print(costs(["gloves"]))`
- Iterate over all keys and print the key and cost of that item

```
for curr_key in costs:
    print(curr_key)
    print(costs[curr_key])
```

Functions

- Block of code that can be called and returns values

```
def my_function(): # declaration
    return True # return value
```

- `my_function() # the call to the function`
- Types of parameters
  - `def my_func(a,b) # requried argument at call`
  - `def my_func(a = 0, b = False) # keyword arguments are assigned default values if not given`
  - Order matters – `def my_func(a, b, c = 0) # keyword arguments are after required ones`
- `my_func(4, 7, c = 2) # calling the function and overriding the c parameter`
- `def my_func(*param)`
  - `*param` is an arbitrary number of arguments that get processed a list
- `def my_func(**kwargs)`
  - `**kwargs` is an arbitrary number of keyword arguments that get processed as a dictionary

Functions Activity

In a python script,

- Create a function: `sum_values()` that uses two parameters that are summed together and returned

  ```
  def sum_values(a, b):
      my_sum = a + b
      return my_sum
  ```

- Call the function with two arguments later in the python script

  ```
  my_ret = sum_values(4, 8)
  print(my_ret)
  ```

- Run the code from the command line
- Change the function to use 3 parameters:
  - `def my_sum(a, b, c)`
  - Call the same way as before
  - `my_sum(4, 8)`
  - Change again to
  - `def my_sum(a, b, c = 0)`
  - Call the same way as before
  - `my_sum(4, 8)`
  - Add some code so that all three variable are summed together `a, b, c`

Classes

Object constructor:

```
class MyClass: # define class
    x = 6

cl = MyClass() # create object from class
print(cl.x) # access property from object
```

`__init__()` Method in Classes

- Build-in by default into every class
- Is called when an object is being created

  ```
  class Person:
      def __init__(self, name, age):
          self.name = name
          self.age = age

  p1 = Person("John", 36)
  p1.name
  ```

Methods in Classes:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def my_method(self):
        print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.my_method()
```