Dictionaries

Table of Contents

Definitions

```
greengrocer = {'fruit' : 'Apple', 'vegetable' : 'Tomato'}
```

```
dict()
```

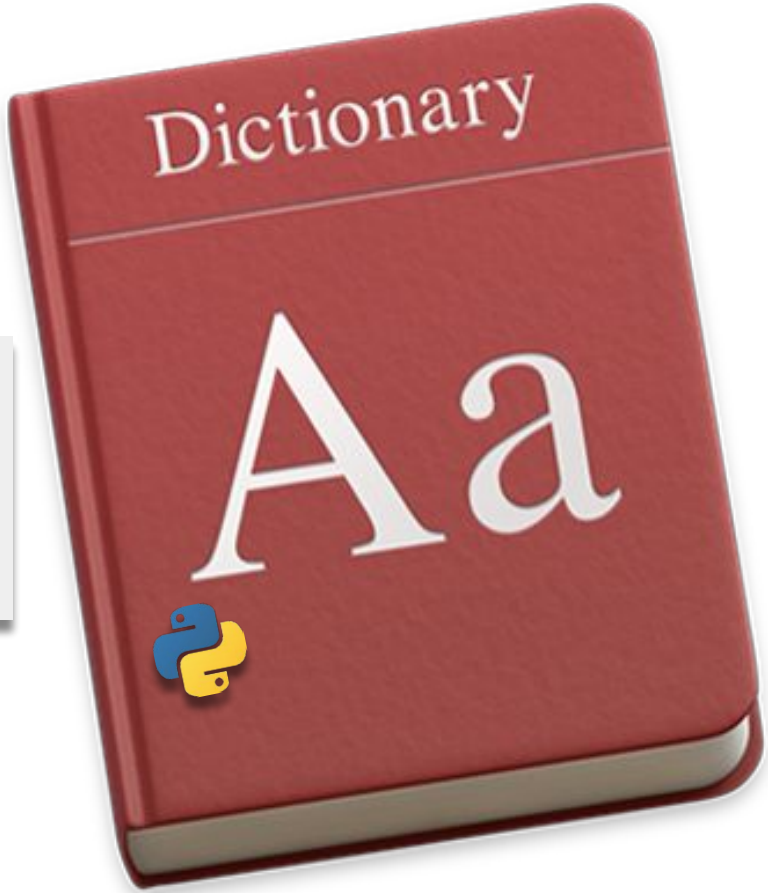
What did you learn from the pre-class content about dictionaries in Python?

Students, write your response!

Definitions

- Dictionaries

```
{key1 : value1,
 key2 : value2}
```



Creating a `dict`

Creating a `dict` (review)

- We have two basic ways to create a dictionary.

- `{}`
- `dict()`

Creating a `dict` (review pre-class)

- Here is an example of simple structure of a `dict`:

```
my_dict = {'key1': 'value1',
           'key2': 'value2',
           'key3': 'value3'
           }

```

Creating a `dict` (review)

- A `dict` can be created by enclosing pairs, separated by commas, in curly-braces 👉 `{}`.
- Another way to create a `dict` is to call the `dict()` function.

```
grocer1 = {'fruit':'apple', 'drink':'water'}
grocer2 = dict(fruit='apple', drink='water')
print(grocer1)
print(grocer2)
```

- `{}`
- `dict()`

What is the output? Try to figure out in your mind...

Creating a `dict` (review)

- A `dict` can be created by enclosing pairs, separated by commas, in curly-braces 👉 `{}`.
- Another way to create a `dict` is to call the `dict()` function.

- `{}`
- `dict()`

```
grocer1 = {'fruit':'apple', 'drink':'water'}
grocer2 = dict(fruit='apple', drink='water')
print(grocer1)
print(grocer2)
```

```
{'fruit': 'apple', 'drink': 'water'}
{'fruit': 'apple', 'drink': 'water'}
```

Creating a `dict`(review pre-class)

- Accessing and assigning an item.

```
state_capitals = {'Arkansas': 'Little Rock',
                  'Colorado': 'Denver',
                  'California': 'Sacramento',
                  'Georgia': 'Atlanta'
                 }

print(state_capitals['Colorado']) # accessing method

```

Creating a `dict` (review pre-class)

- Assigning a value to a key

```python
state_capitals = {'Arkansas': 'Little Rock',
                  'Colorado': 'Denver',
                  'California': 'Sacramento',
                  'Georgia': 'Atlanta'
                 }

print(state_capitals['Colorado']) # accessing method

```

```
Denver

```

Creating a `dict` (review pre-class)

- Let's add a new item into the `dict`.

```
state_capitals = {'Arkansas': 'Little Rock',
                  'Colorado': 'Denver',
                  'California': 'Sacramento',
                  'Georgia': 'Atlanta'
                 }

state_capitals['Virginia'] = 'Richmond' # adding a new item

print(state_capitals)

```

Creating a `dict` (review pre-class)

- Let's add a new item into the `dict`.

```python
state_capitals = {'Arkansas': 'Little Rock',
                  'Colorado': 'Denver',
                  'California': 'Sacramento',
                  'Georgia': 'Atlanta'
                 }

state_capitals['Virginia'] = 'Richmond' # adding a new item

print(state_capitals)

```

```
{'Arkansas': 'Little Rock',
'Colorado': 'Denver',
'California': 'Sacramento',
'Georgia': 'Atlanta',
'Virginia': 'Richmond'}

```

Creating a `dict` (review pre-class)

Tips:

- Note that keys and values can be of different types.

```
mix_values = {'animal': ('dog', 'cat'),  # tuple type
              'planet': ['Neptun', 'Saturn', 'Jupiter'],  # list type
              'number': 40,  # int type
              'pi': 3.14,  # float type
              'is_good': True}  # bool type

mix_keys = {22 : "integer",
            1.2 : "float",
            True : "boolean",
            "key" : "string"}

```

Creating a `dict`

- **Task** 👇
  - Let's create a `dict` (named `family`) which consists of names of 3 members of your family.
  - Each person should have only the first **names.**
  - For example;

    name1

    name2

    •

    •

Create using curly braces 👉 `{}`

Creating a `dict`

- The code can be like :

```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha'
         }
```

Creating a `dict`

- **Task** 👇
  - Add a new family member name to the dictionary you created.

Creating a `dict`

- The code can be like :

```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha'
         }
family['name4'] = 'Tom'
print(family)
```

```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha',
          'name4': 'Tom'
         }
```

Creating a `dict`

- Now, it's time to create a `dict` using `dict()` function :

```
dict_by_dict = dict(animal='dog', planet='neptun', number=40, pi=3.14, is_good=True)

print(dict_by_dict)

```

Creating a `dict`

- Now, it's time to create a `dict` using `dict()` function :

```python
dict_by_dict = dict(animal='dog', planet='neptun', number=40, pi=3.14, is_good=True)

print(dict_by_dict)

```

```
{'animal': 'dog',
'planet': 'neptun',
'number': 40,
'pi': 3.14,
'is_good': True}

```

Creating a `dict`

- Now, it's time to create a `dict` using `dict()` function :

```python
dict_by_dict = dict(animal='dog', planet='neptun', number=40, pi=3.14, is_good=True)

print(dict_by_dict)

```

```
{'animal': 'dog',
'planet': 'neptun',
'number': 40,
'pi': 3.14,
'is_good': True}

```

⚠ **Avoid ! :**

- Do not use quotes for `keys` when using the `dict()` function to create a dictionary.

Creating a `dict`

- **Task** 👇
  - Create the same `dict` using `dict()` function.

Creating a `dict`

- The code can be like :

```
family = dict(name1 = 'Joseph', name2 = 'Bella', name3 = 'Aisha',
name4 = 'Tom')

print(family)
```

```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha',
          'name4': 'Tom'
         }
```

Main Operations with Dictionaries

Main Operations with `dicts` (review)

clear()

Remove all items from the dictionary.

copy()

Return a shallow copy of the dictionary.

classmethod **fromkeys**(*iterable*[, *value*])

Create a new dictionary with keys from *iterable* and values set to *value*.

`fromkeys()` is a class method that returns a new dictionary. *value* defaults to `None`. All of the values refer to just a single instance, so it generally doesn't make sense for *value* to be a mutable object such as an empty list. To get distinct values, use a dict comprehension instead.

get(*key*[, *default*])

Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

items()

Return a new view of the dictionary's items (`(key, value)` pairs). See the documentation of view objects.

keys()

Return a new view of the dictionary's keys. See the documentation of view objects.

pop(*key*[, *default*])

If *key* is in the dictionary, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.

popitem()

Remove and return a `(key, value)` pair from the dictionary. Pairs are returned in LIFO order.

`popitem()` is useful to destructively iterate over a dictionary, as often used in set algorithms. If the dictionary is empty, calling `popitem()` raises a `KeyError`.

Changed in version 3.7: LIFO order is now guaranteed. In prior versions, `popitem()` would return an arbitrary key/value pair.

reversed(d)

Return a reverse iterator over the keys of the dictionary. This is a shortcut for `reversed(d.keys())`.

New in version 3.8.

setdefault(*key*[, *default*])

If *key* is in the dictionary, return its value. If not, insert *key* with a value of *default* and return *default*. *default* defaults to `None`.

update([*other*])

Update the dictionary with the key/value pairs from *other*, overwriting existing keys. Return `None`.

`update()` accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

values()

Return a new view of the dictionary's values. See the documentation of view objects.

An equality comparison between one `dict.values()` view and another will always return `False`. This also applies when comparing `dict.values()` to itself:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

https://docs.python.org/3/library/stdtypes.html#dict-views

Main Operations with `dicts` (review)

- You can access all;
  - `items` using the `.items()` method,
  - `keys` using the `.keys()` method,
  - `values` using the `.values()` method.

Main Operations with `dicts` (review)

- Let's take a look at this example :

```
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True}

print(dict_by_dict.items(), '\n')
print(dict_by_dict.keys(), '\n')
print(dict_by_dict.values())
```

What is the output? Try to figure out in your mind...

Main Operations with `dicts` (review)

- Let's take a look at this example :

```python
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True}

print(dict_by_dict.items(), '\n')
print(dict_by_dict.keys(), '\n')
print(dict_by_dict.values())

```

```
dict_items([('animal', 'dog'), ('planet', 'neptun'),
            ('number', 40), ('pi', 3.14), ('is_good', True)])

dict_keys(['animal', 'planet', 'number', 'pi', 'is_good'])

dict_values(['dog', 'neptun', 40, 3.14, True])

```

Main Operations with `dicts`

- **Task** 👇
  - Access and print the **`items`**, **`keys`** and **`values`** of the same `family` `dict` you created.
  - Note : Get the output of the above as a `list` type.

Main Operations with `dicts`

- The code can be like :

```
print(list(family.items()), "\n")
print(list(family.keys()), "\n")
print(list(family.values()))
```

```
[('name1', 'Joseph'), ('name2', 'Bella'), ('name3', 'Aisha'), ('name4', 'Tom')]

['name1', 'name2', 'name3', 'name4']

['Joseph', 'Bella', 'Aisha', 'Tom']
```

Main Operations with Dictionaries

Main Operations with `dicts` (review)

- `.update()` method :

```
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True}

dict_by_dict.update({'is_bad': False})

print(dict_by_dict)

```

Main Operations with `dicts` (review)

- Another way to add a new **item** into a `dict` is the `.update()` method.

```
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True}

dict_by_dict.update({'is_bad': False})

print(dict_by_dict)

```

```
{'animal': 'dog',
'planet': 'neptun',
'number': 40,
'pi': 3.14,
'is_good': True,
'is_bad': False}

```

Main Operations with `dicts`

- The code can be like :

```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha',
          'name4': 'Tom'
         }
family.update({'name5': 'Alfred', 'name6': 'Ala'})
print(family)
```

```
family ={'name1': 'Joseph',
         'name2': 'Bella',
         'name3': 'Aisha',
         'name4': 'Tom',
         'name5': 'Alfred',
         'name6': 'Ala'}
```

Main Operations with `dicts` (review)

- Python allows us to remove an **item** from a `dict` using the `del` function.

The formula syntax is : `del dictionary_name['key']`

```
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True,
                'is_bad': False}

del dict_by_dict['animal']

print(dict_by_dict)

```

Main Operations with `dicts` (review)

- Python allows us to remove an **item** from a `dict` using the `del` function.

The formula syntax is : `del dictionary_name['key']`

```python
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True,
                'is_bad': False}

del dict_by_dict['animal']

print(dict_by_dict)

```

```
{'planet': 'neptun',
'number': 40,
'pi': 3.14,
'is_good': True,
'is_bad': False}

```

Main Operations with `dicts`

- The code can be like :

```
del family['name2']
del family['name3']

print(family)
```

```
family = {'name1': 'Joseph',
          'name4': 'Tom',
          'name5': 'Alfred'
         }
```

Main Operations with `dicts`

- The code can be like :

```
del family['name2']
del family['name3']

print(family)
```

Can you do the same thing in a single line ❓

```
family = {'name1': 'Joseph',
          'name4': 'Tom',
          'name5': 'Alfred'
         }
```

Main Operations with `dicts`

- The code can be like :

Option-1

```
del family['name2']
del family['name3']

print(family)
```

Option-2

```
del family['name2'], family['name3']

print(family)
```

```
family = {'name1': 'Joseph',
          'name4': 'Tom',
          'name5': 'Alfred'
         }
```

Main Operations with `dicts`

- The code can be like :

Option-3

```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha',
          'name4': 'Tom'
         }
# using pop to return and remove key-value pair.
pop_ele = family.pop('name1')
print("deleted..:", pop_ele)
print(family)
```

```
deleted..: Joseph
{'name2': 'Bella', 'name3': 'Aisha', 'name4': 'Tom'}
```

Main Operations with `dicts`

- **If the key is not present in the dictionary, it raises a KeyError.**

KeyError

```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha',
          'name4': 'Tom'
         }
>>> family.pop('name5')
## or
>>> del family['name5']
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
```

Main Operations with `dicts`

- **If the key is not present in the dictionary, it raises a KeyError.**

KeyError Solution?

```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha',
          'name4': 'Tom'
         }
>>> family.pop('name5', 'absent in the dict.')
```

message

```
'absent in the dict.'
```

Main Operations with dicts

- # How to delete multiple values?

Option-1

```
family = list(family.items())    ## convert to list
del family[0:2]                  ## slicing
print(dict(family))              ## convert to dict
```

Option-2

```
keys = ['name1', 'name2', 'name3']
## Or can be deleted in a loop.
for key in keys:
    del family[key]
print(family)
```

```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha',
          'name4': 'Tom',
          'name5': 'Ala'
         }
```

```
{'name4': 'Tom', 'name5': 'Ala'}
```

Main Operations with `dicts`

`popitem`(): Remove and return a `(key, value)` pair from the dictionary. Pairs are returned in **LIFO** order.

```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha',
          'name4': 'Tom'
         }
print(family.popitem() )
```

Main Operations with `dicts`

Remove and return a `(key, value)` pair from the dictionary. Pairs are returned in **LIFO** order.

```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha',
          'name4': 'Tom'
         }
print(family.popitem() )
```

```
('name4', 'Tom')
```

Nested Dictionaries

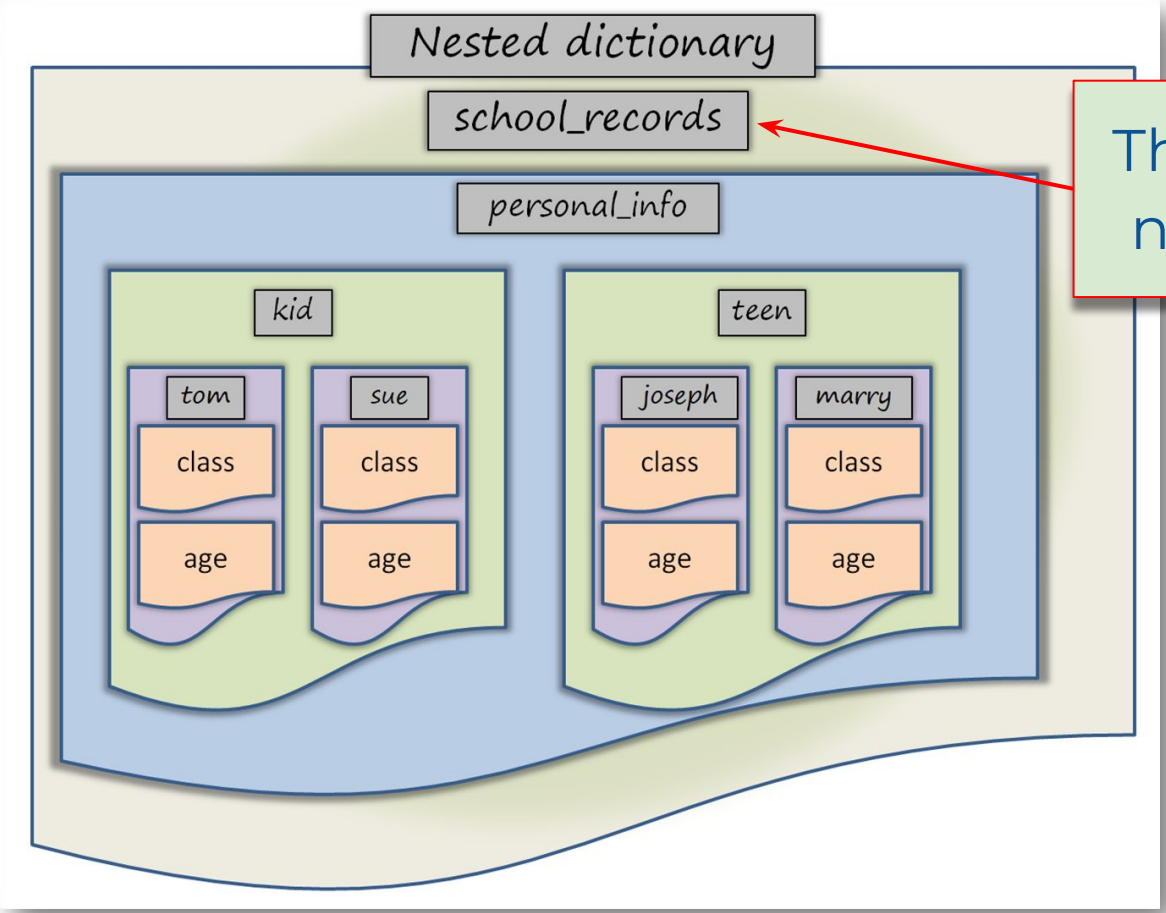
Nested `dicts` (review pre-class)

- In some cases you need to work with nested `dict`. Consider the following pre-class example :

```
school_records={
    "personal_info":
        {"kid":{"tom": {"class": "intermediate", "age": 10},
                "sue": {"class": "elementary", "age": 8}
                },
         "teen":{"joseph":{"class": "college", "age": 19},
                 "marry":{"class": "high school", "age": 16}
                },
        },

    "grades_info":
        {"kid":{"tom": {"math": 88, "speech": 69},
                "sue": {"math": 90, "speech": 81}
                },
         "teen":{"joseph":{"coding": 80, "math": 89},
                 "marry":{"coding": 70, "math": 96}
                },
        },
}
```

Nested `dicts` (review pre-class)



The first part of the nested dictionary.

Nested `dicts` (review pre-class)

- You can use traditional accessing method - square brackets - also in the nested dictionaries.

```
school_records={
    "personal_info":
        {"kid":{"tom": {"class":"intermediate", "age":10},
                "sue": {"class":"elementary", "age":8}
               },
         "teen":{"joseph":{"class":"college", "age":19},
                "marry":{"class":"high school", "age":16}
               },
        },
}

print(school_records['personal_info']['teen']['marry']['age'])
```

Nested `dicts` (review pre-class)

- You can use traditional accessing method - square brackets - also in the nested dictionaries.

```
school_records={
    "personal_info":
        {"kid":{"tom": {"class":"intermediate", "age":10},
                "sue": {"class":"elementary", "age":8}
               },
         "teen":{"joseph":{"class":"college", "age":19},
                "marry":{"class":"high school", "age":16}
               },
        },
}

print(school_records['personal_info']['teen']['marry']['age'])
```

```
16
```

Nested `dicts`

- **Task** : Access and print the **exams** and their **grades** of **Joseph** as in two types; one is a `list` form and one is a `dict`.

```
school_records={
    "personal_info":
        {"kid":{"tom": {"class": "intermediate", "age": 10},
                "sue": {"class": "elementary", "age": 8}
               },
         "teen":{"joseph":{"class": "college", "age": 19},
                 "marry":{"class": "high school", "age": 16}
               },
        },

    "grades_info":
        {"kid":{"tom": {"math": 88, "speech": 69},
                "sue": {"math": 90, "speech": 81}
               },
         "teen":{"joseph":{"coding": 80, "math": 89},
                 "marry":{"coding": 70, "math": 96}
               },
        },
}
```

Students, write your response!

Nested dicts

- The code can be like :

```
school_records={
    "personal_info":
        {"kid":{"tom": {"class": "intermediate", "age": 10},
                "sue": {"class": "elementary", "age": 8}
                },
         "teen":{"joseph":{"class": "college", "age": 19},
                 "marry":{"class": "high school", "age": 16}
                },
        },

    "grades_info":
        {"kid":{"tom": {"math": 88, "speech": 69},
                "sue": {"math": 90, "speech": 81}
                },
         "teen":{"joseph":{"coding": 80, "math": 89},
                 "marry":{"coding": 70, "math": 96}
                },
        },
}
print(list(school_records["grades_info"]["teen"]["joseph"].items()))
print(school_records["grades_info"]["teen"]["joseph"])
```

Output

```
[('coding', 80), ('math', 89)]
{'coding': 80, 'math': 89}
```

Nested `dicts`

- **Task** 👇
  - Let's create and print a `dict` (named `friends`) which consists of **first** and **last** names of your friends.
  - Each person should have first and last **names.**
  - For example;
    friend1: (first : Sue, last : Bold)
    friend2: (first : Steve, last : Smith)
    •
    •

Create using curly braces 👉 `{}`

Nested `dicts`

- The code can be like :

```
friends = {
    "friend1" : {"first" : "Sue", "last" : "Bold"},
    "friend2" : {"first" : "Steve", "last" : "Smith"},
    "friend3" : {"first" : "Sergio", "last" : "Tatoo"}
}
print(friends)
```

Nested `dicts`

Create using curly braces 👉 `{}`

- **Task** 👇
  - Let's create and print a `dict` (named `favourite`) which consists of first and last names of your **friends** and **family** members.
  - Each person should have first and last names and the groups (friends and family) have three person each.
  - For example;

```
friends :
    friend1: (first  : Sue, last : Bold)
family :
    family1:      (first      :      Steve,      last      :      Smith)
```

Nested dicts

- The code can be like :

```
1  favourite = {
2      "friends" : {
3          "friend1" : {"first" : "Sue", "last" : "Bold"},
4          "friend2" : {"first" : "Steve", "last" : "Smith"},
5          "friend3" : {"first" : "Sergio", "last" : "Tatoo"}
6          },
7      "family" : {
8          "family1" : {"first" : "Mary", "last" : "Tisa"},
9          "family2" : {"first" : "Samuel", "last" : "Brown"},
10         "family3" : {"first" : "Tom", "last" : "Happy"}
11         }
12 }
13 print(favourite)
14
```

Nested `dicts`

- What *statement* will remove the entry in the dictionary for key `'family3'`?

```
1  favourite = {
2      "friends" : {
3          "friend1" : {"first" : "Sue", "last" : "Bold"},
4          "friend2" : {"first" : "Steve", "last" : "Smith"},
5          "friend3" : {"first" : "Sergio", "last" : "Tatoo"}
6          },
7      "family" : {
8          "family1" : {"first" : "Mary", "last" : "Tisa"},
9          "family2" : {"first" : "Samuel", "last" : "Brown"},
10         "family3" : {"first" : "Tom", "last" : "Happy"}
11         }
12 }
13 print(favourite)
14
```

Nested `dicts`

- What *statement* will **remove** the entry in the dictionary for key `'family3'`?

```
1  favourite = {
2      "friends" : {
3          "friend1" : {"first" : "Sue", "last" : "Bold"},
4          "friend2" : {"first" : "Steve", "last" : "Smith"},
5          "friend3" : {"first" : "Sergio", "last" : "Tatoo"}
6          },
7      "family" : {
8          "family1" : {"first" : "Mary", "last" : "Tisa"},
9          "family2" : {"first" : "Samuel", "last" : "Brown"},
10         "family3" : {"first" : "Tom", "last" : "Happy"}
11         }
12 }
```

```
del_family = favourite['family'].pop('family3')
print(del_family)
```

Nested `collections`

- What is the expression involving **`y`** that **access**es the value `20`?

```
dt = [
    'a',
    'b',
    {
        'foo': 1,
        'bar':
        {
            'x' : 10,
            'y' : 20,
            'z' : 30
        },
        'baz': 3
    },
    'c',
    'd',
    'e'
]
```

Nested `collections`

- What is the expression involving `y` that accesses the value `20`?

```
dt = [
    'a',
    'b',
    {
        'foo': 1,
        'bar':
        {
            'x' : 10,
            'y' : 20,
            'z' : 30
        },
        'baz': 3
    },
    'c',
    'd',
    'e'
]
dt[2]['bar']['y']
```

[20] ✓ 0.7s

... 20