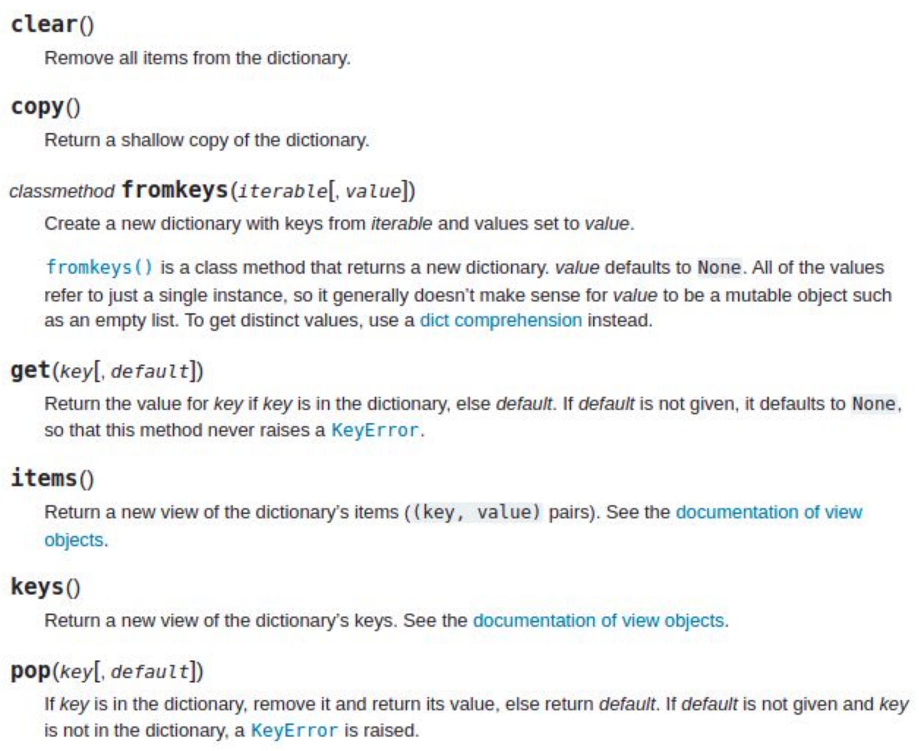
Main Operations with Dictionaries

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`()

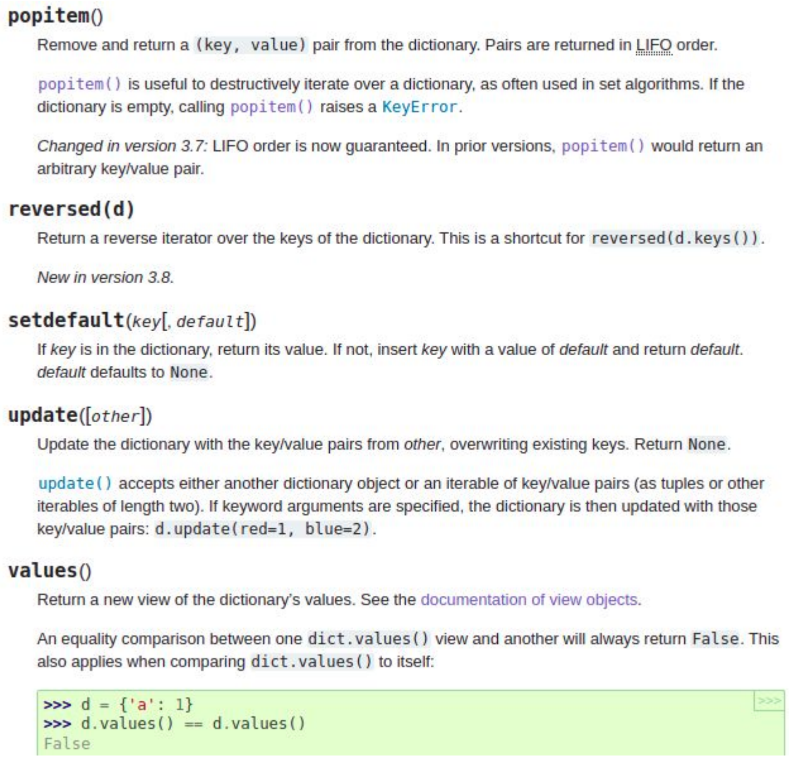
Return a new view of the dictionary's items (`(key, value)` pairs). See the documentation of view objects.

`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.

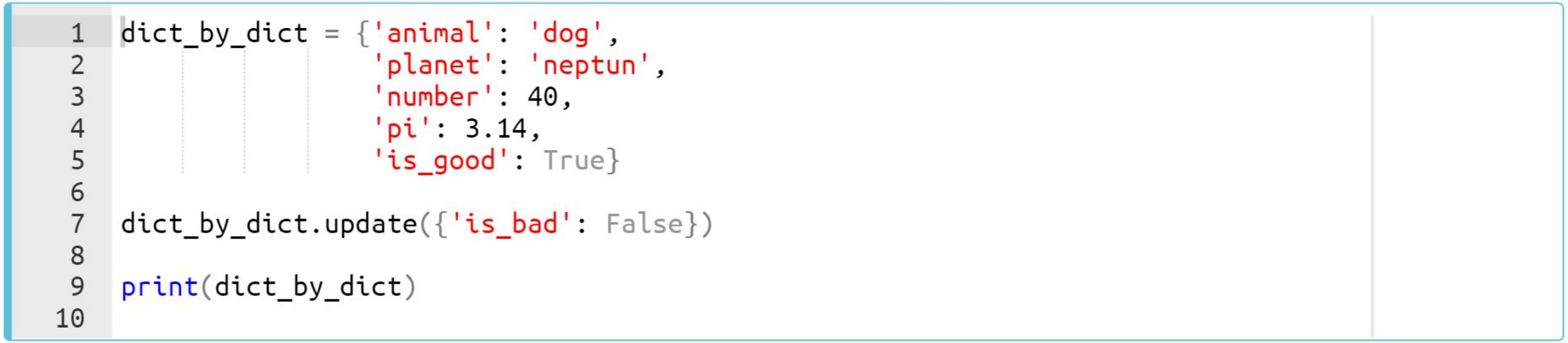
An equality comparison between one `dict.values()` view and another will always return `False`. This also applies when comparing `dict.values()` to itself:

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

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

Main Operations with `dicts` (review)

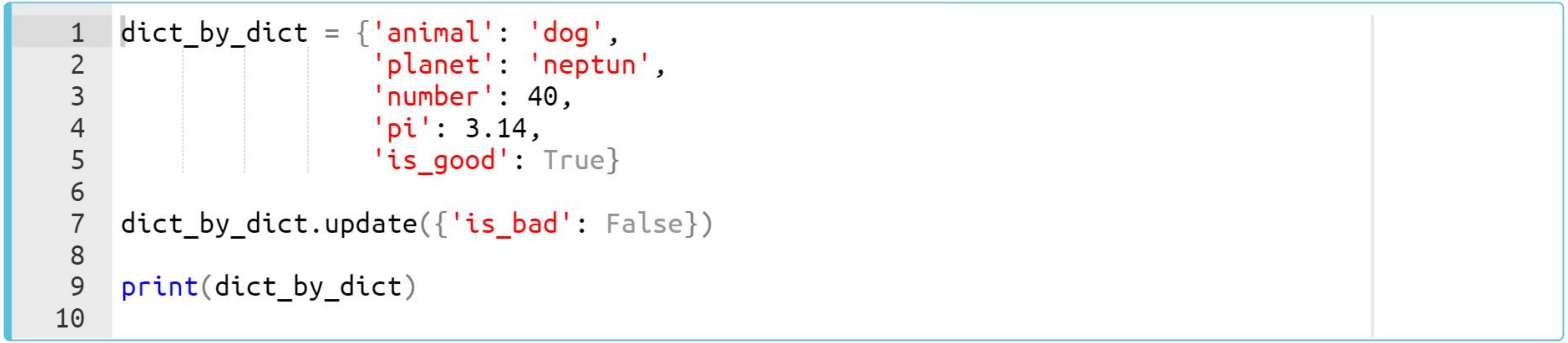
- `.update()` method :



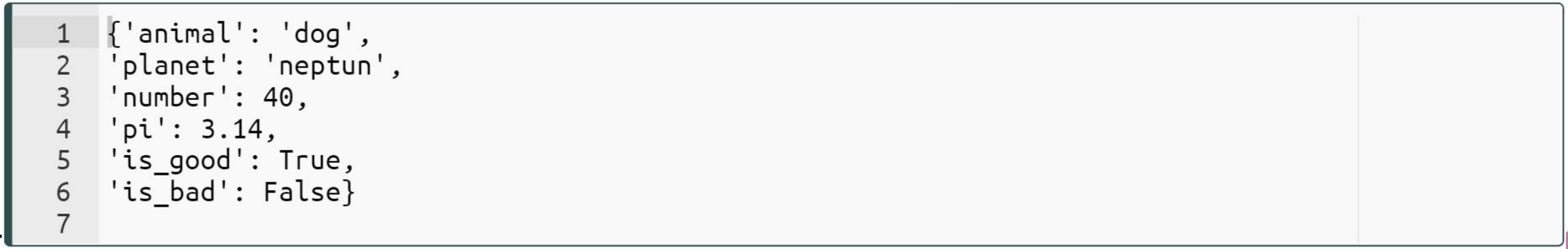
```python
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True}

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.



```python
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}
```

CLARUSWAY©
WAY TO REINVENT YOURSELF

Main Operations with `dicts`

- The code can be like :

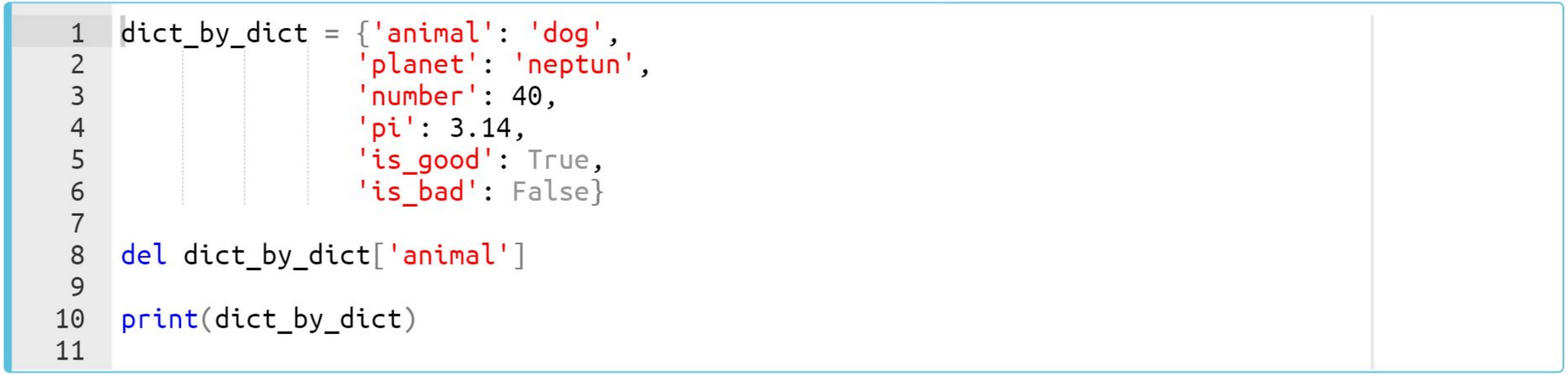
```python
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha',
          'name4': 'Tom'
         }
family.update({'name5': 'Alfred', 'name6': 'Ala'})
print(family)
```

```
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']`

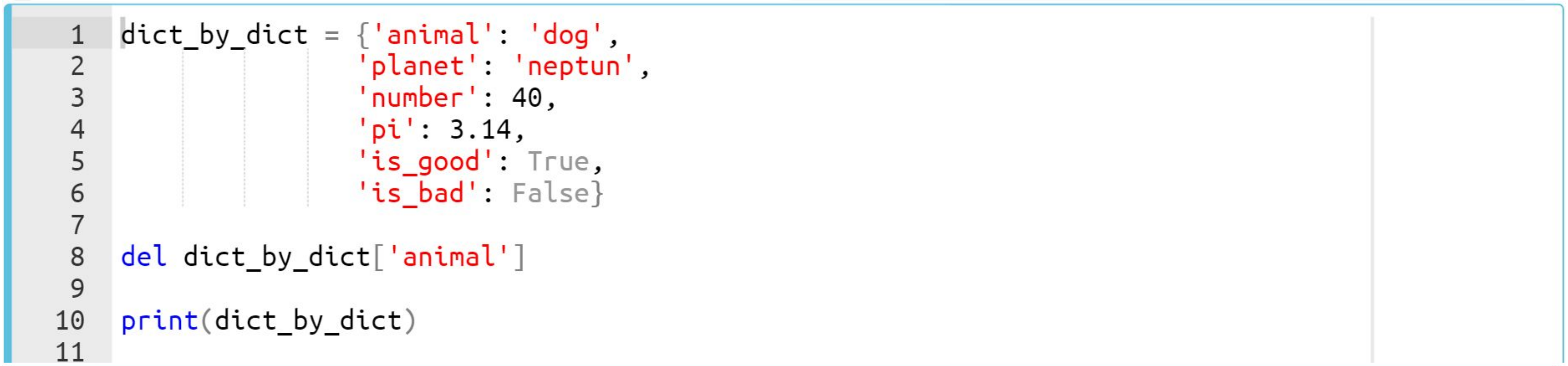


```python
dict_by_dict = {'animal': 'dog',
                'planet': 'neptun',
                'number': 40,
                'pi': 3.14,
                'is_good': True,
                'is_bad': False}

del dict_by_dict['animal']

print(dict_by_dict)
```

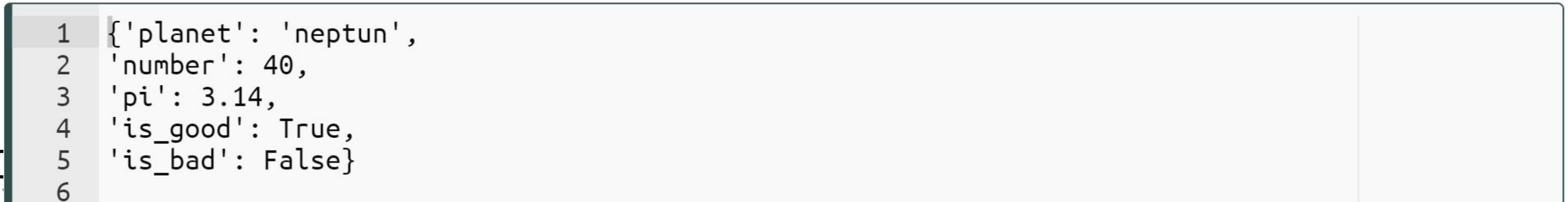
Main Operations with `dicts` (review)

- 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 :

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

print(family)
```

Can you do the same thing in a single line ❓

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

Students, write your response!

Main Operations with `dicts`

- The code can be like :

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

print(family)
```

Option-1

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

print(family)
```

Option-2

```python
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.**

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

KeyError

```
---------------------------------------------------------------------------
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)
```

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

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

CLARUSWAY©
WAY TO REINVENT YOURSELF

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() )
```

CLARUSWAY©
WAY TO REINVENT YOURSELF

Main Operations with `dicts`

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

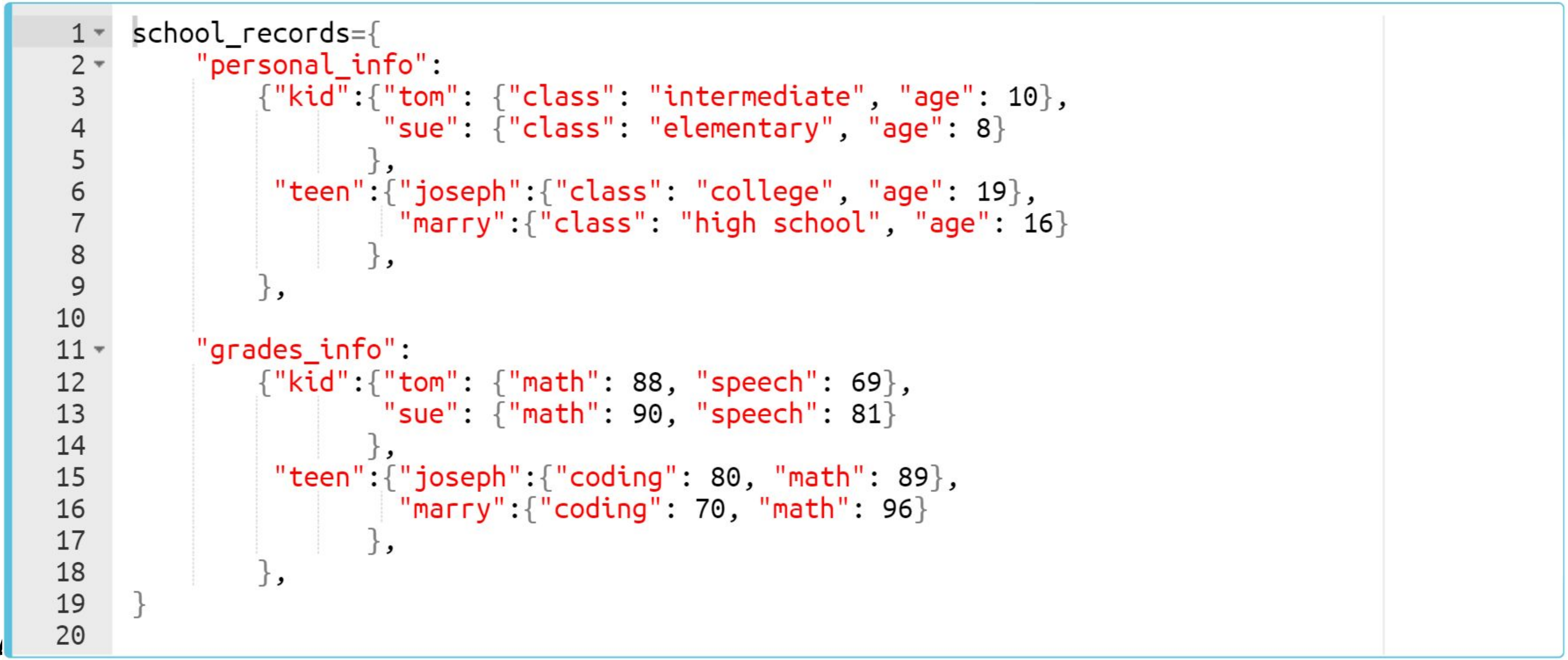
```
family = {'name1': 'Joseph',
          'name2': 'Bella',
          'name3': 'Aisha',
          'name4': 'Tom'
         }
print(family.popitem() )
```

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

Nested Dictionaries

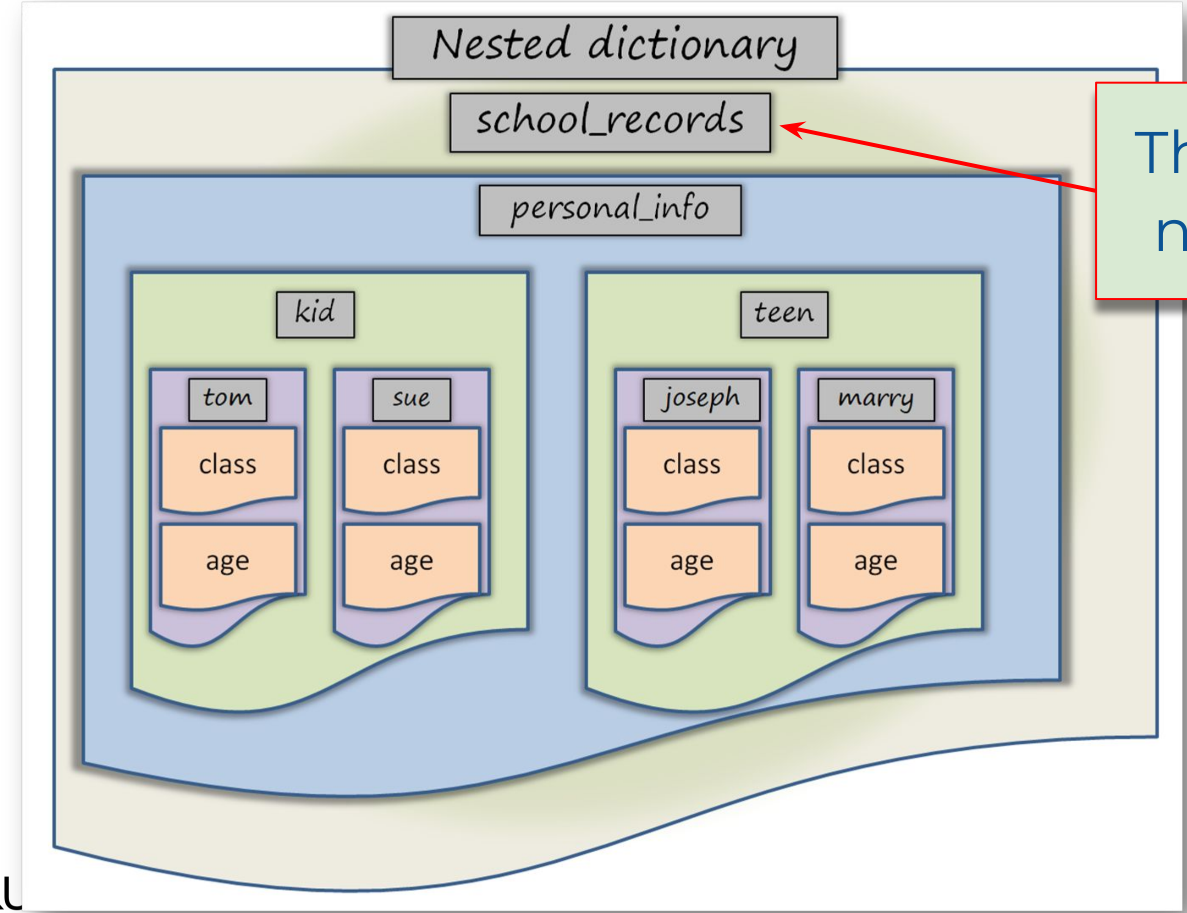
Nested `dicts` (review pre-class)

- 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}
                      },
                },
}
```

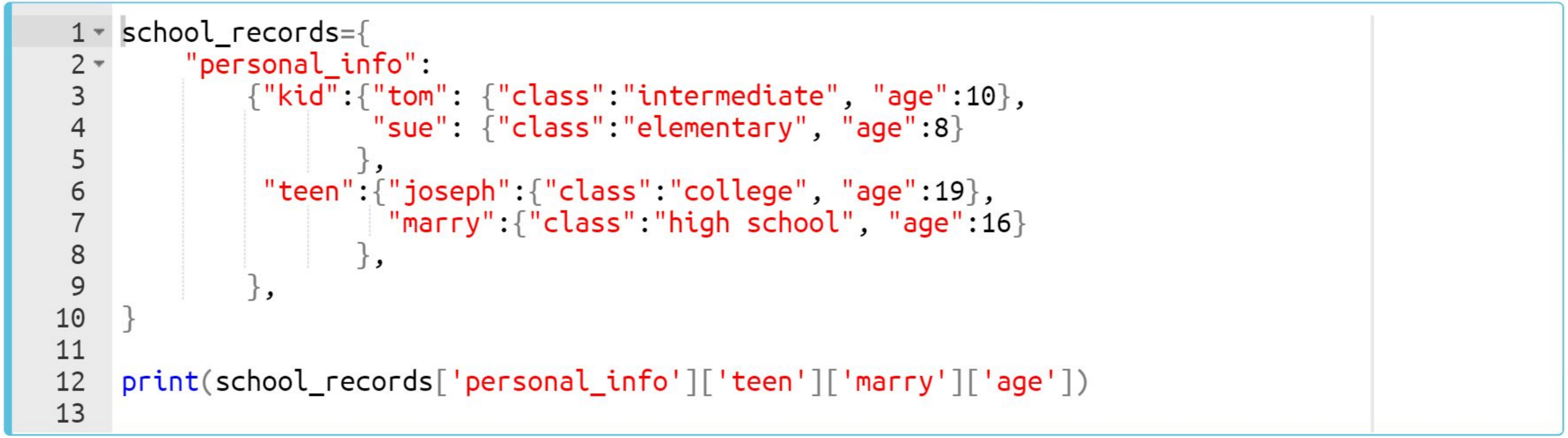
Nested `dicts` (review pre-class)



The first part of the nested dictionary.

Nested `dicts` (review pre-class)

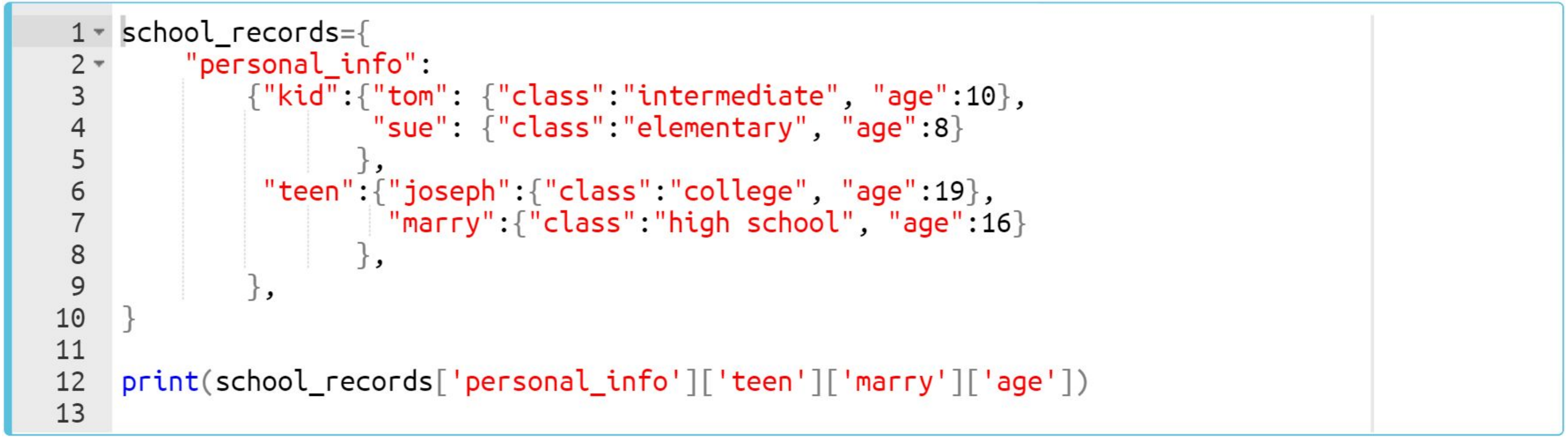
- You can use traditional accessing method **-** square brackets **-** also in the nested dictionaries.



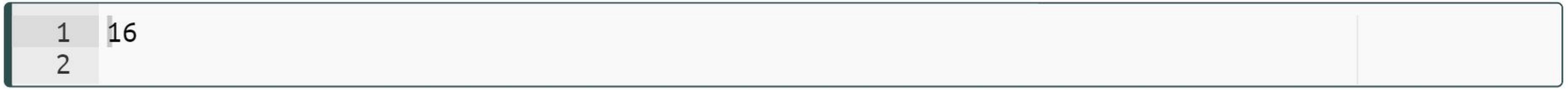
```
school_records={
    "personal_info":
        {"kid":{"tom": {"class":"intermediate", "age":10},
                "sue": {"class":"elementary", "age":8}
               },
         "teen":{"joseph":{"class":"college", "age":19},
                 "marry":{"class":"high school", "age":16}
          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

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.



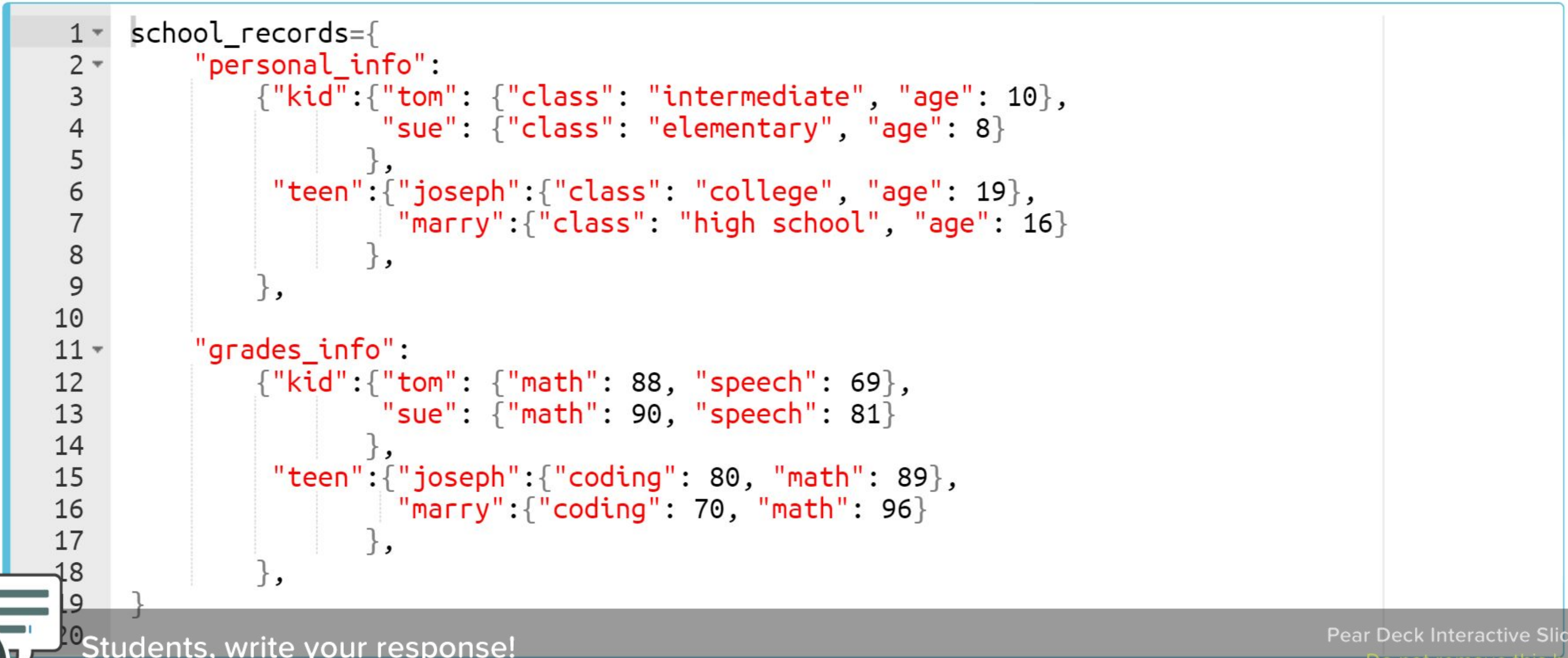
```
school_records={
    "personal_info":
        {"kid":{"tom": {"class":"intermediate", "age":10},
                "sue": {"class":"elementary", "age":8}
               },
         "teen":{"joseph":{"class":"college", "age":19},
                 "marry":{"class":"high school", "age":16}
          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

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`.

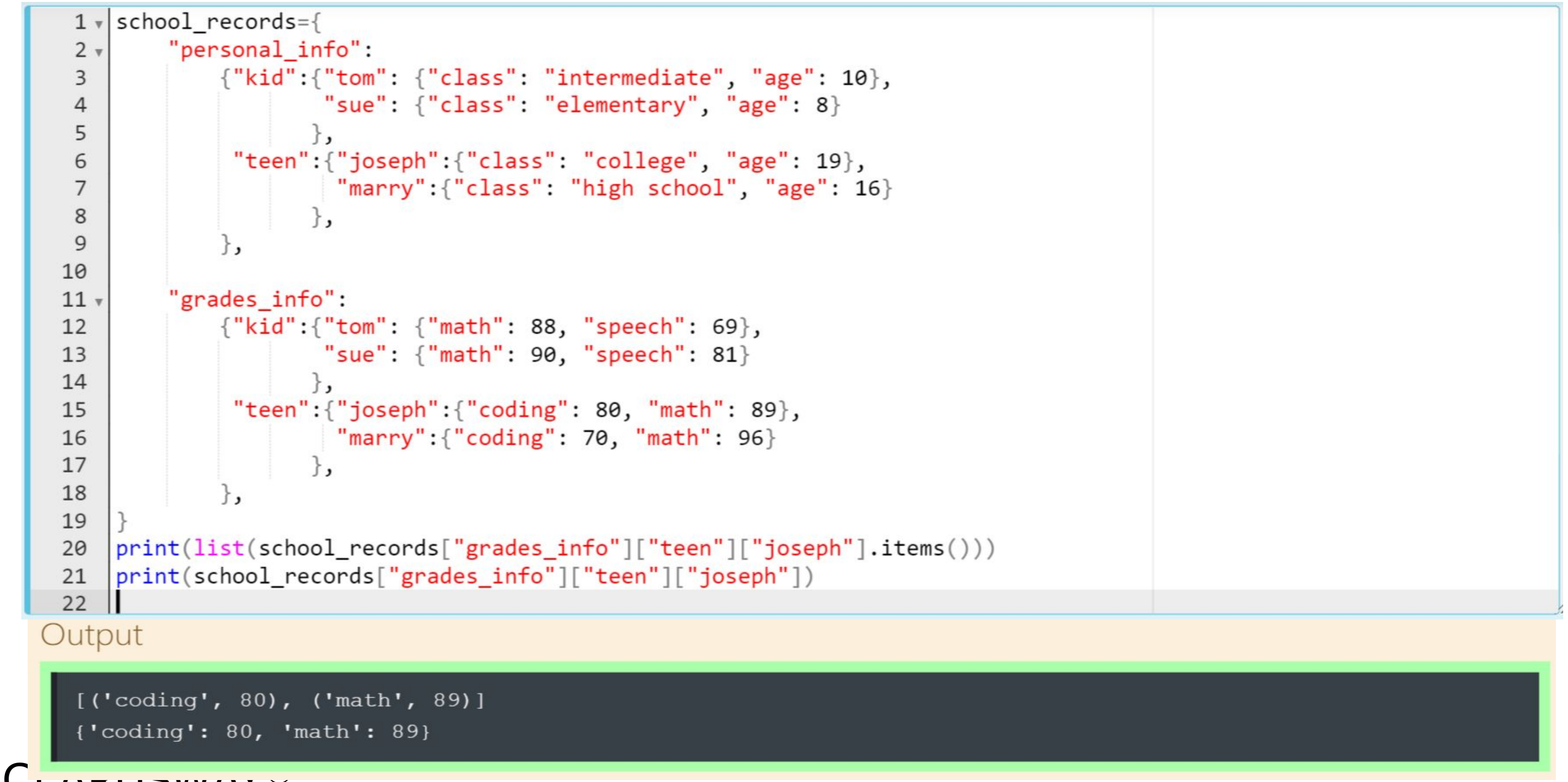


```
school_records={
    "personal_info":
        {"kid":{"tom": {"class": "intermediate", "age": 10},
                "sue": {"class": "elementary", "age": 8}
               },
         "teen":{"joseph":{"class": "college", "age": 19},
                 "marry":{"class": "high school", "age": 16}
  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

    "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}
```

CLARUSWAY
WAY TO REINVENT YOURSELF

Nested `dicts`

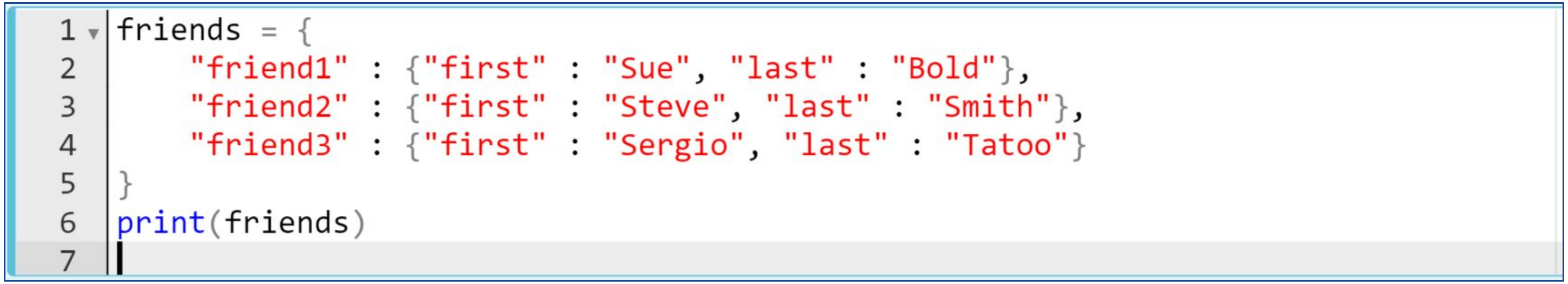
- **Task** 👇
  - Let's create and print a `dict` (named `friends`) which consists of **first** and **last** names of your friends.
  - Each person should have first and last **names.**
  - For example;
    friend1: (first : Sue, last : Bold)
    friend2: (first : Steve, last : Smith)
    -
    -

Create using curly braces 👉`{}`

CLARUSWAY©
WAY TO REINVENT YOURSELF

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`

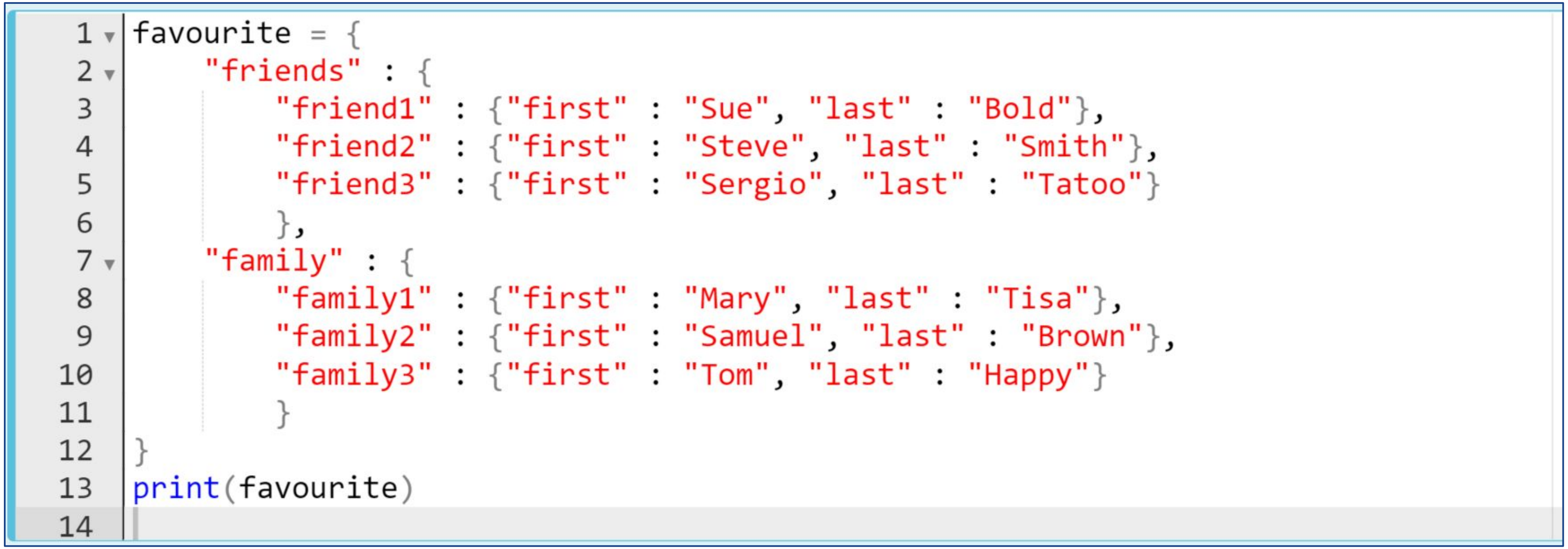
Create using curly braces 👉{}

- **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`

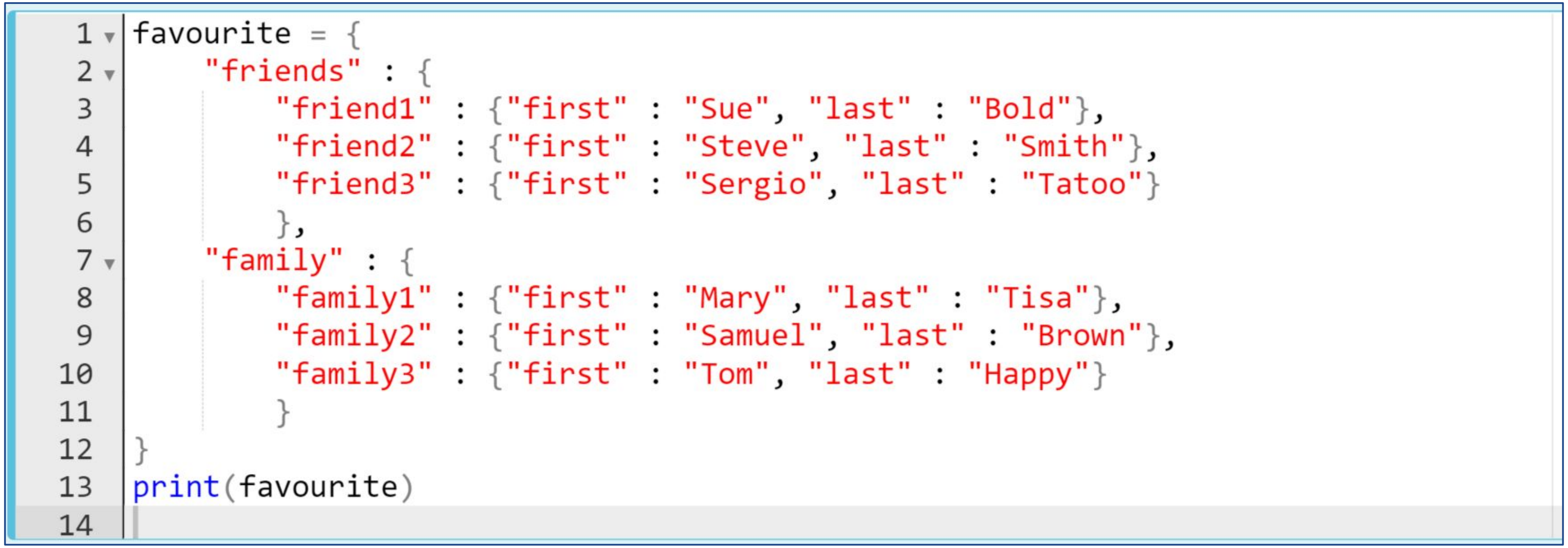
- The code can be like :



```python
favourite = {
         "friends" : {
                  "friend1" : {"first" : "Sue", "last" : "Bold"},
                  "friend2" : {"first" : "Steve", "last" : "Smith"},
                  "friend3" : {"first" : "Sergio", "last" : "Tatoo"}
                  },
         "family" : {
                  "family1" : {"first" : "Mary", "last" : "Tisa"},
                  "family2" : {"first" : "Samuel", "last" : "Brown"},
                  "family3" : {"first" : "Tom", "last" : "Happy"}
                  }
}
print(favourite)

```

Nested `dicts`

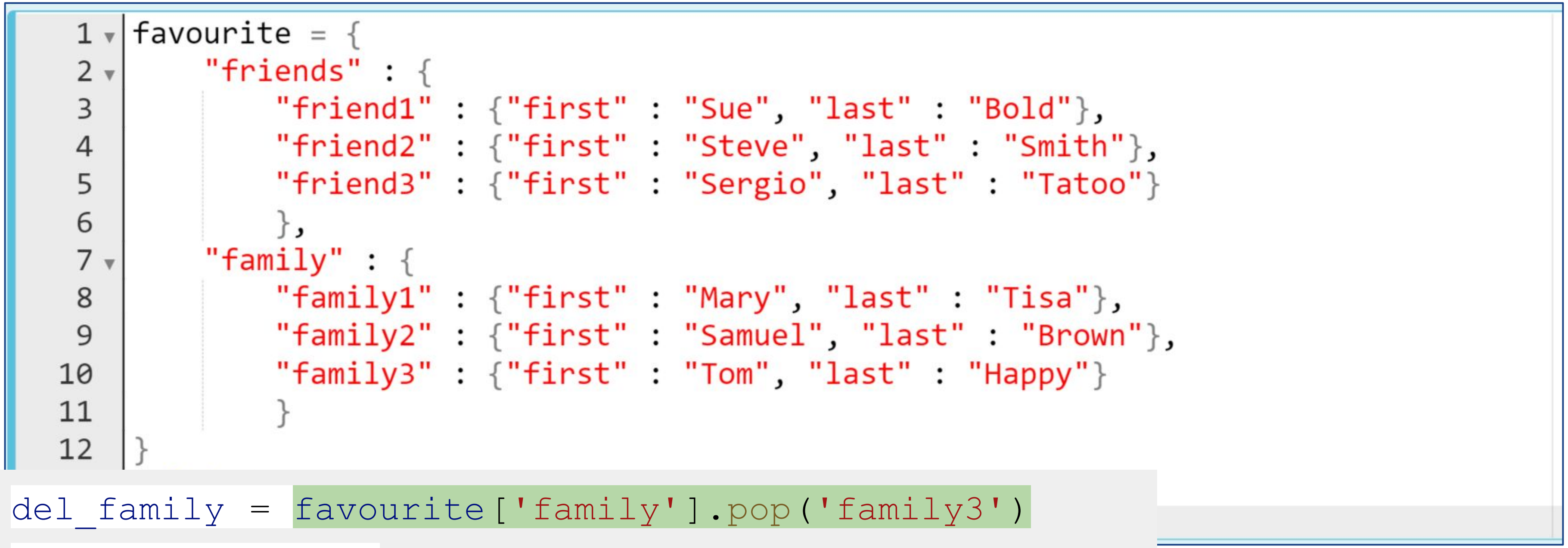
- What *statement* will remove the entry in the dictionary for key `'family3'`?



```python
favourite = {
          "friends" : {
                "friend1" : {"first" : "Sue", "last" : "Bold"},
                "friend2" : {"first" : "Steve", "last" : "Smith"},
                "friend3" : {"first" : "Sergio", "last" : "Tatoo"}
                },
          "family" : {
                "family1" : {"first" : "Mary", "last" : "Tisa"},
                "family2" : {"first" : "Samuel", "last" : "Brown"},
                "family3" : {"first" : "Tom", "last" : "Happy"}
                }
}
print(favourite)
```

Nested `dicts`

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



```
favourite = {
        "friends" : {
            "friend1" : {"first" : "Sue", "last" : "Bold"},
            "friend2" : {"first" : "Steve", "last" : "Smith"},
            "friend3" : {"first" : "Sergio", "last" : "Tatoo"}
            },
        "family" : {
            "family1" : {"first" : "Mary", "last" : "Tisa"},
            "family2" : {"first" : "Samuel", "last" : "Brown"},
            "family3" : {"first" : "Tom", "last" : "Happy"}
            }
}
```

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

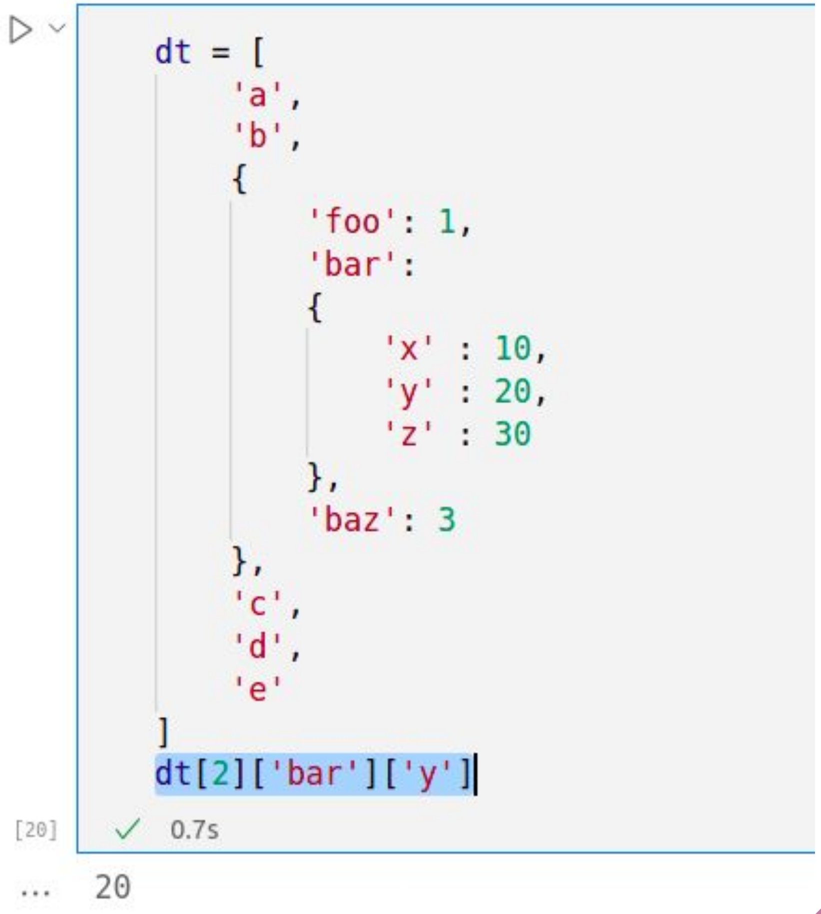
Nested collections

- 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`?



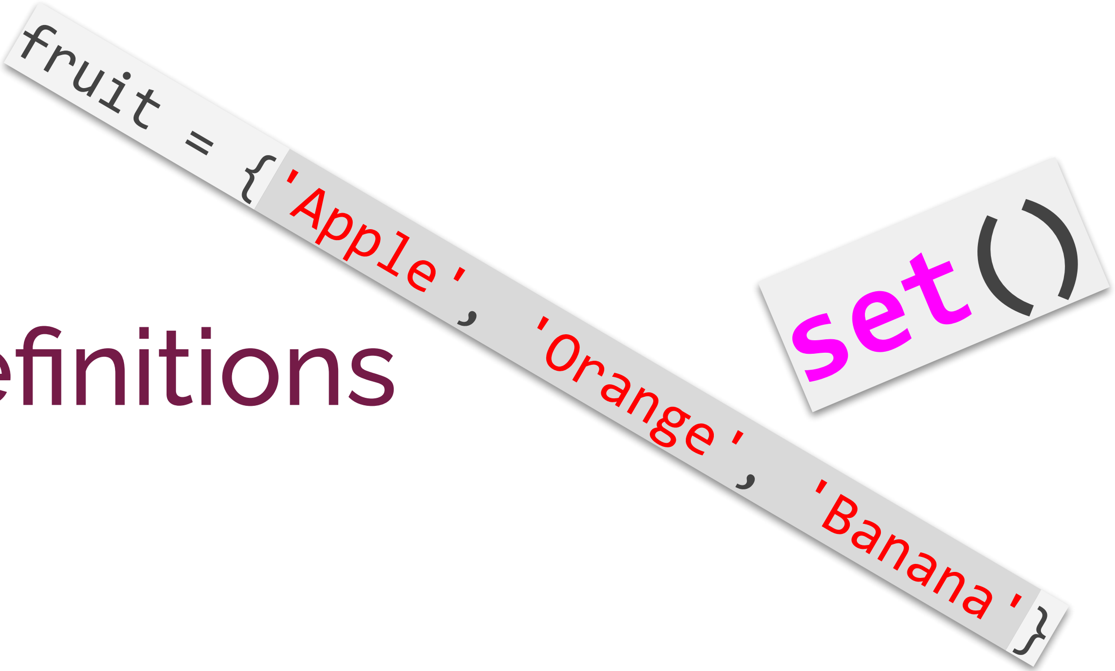
```
dt = [
    'a',
    'b',
    {
        'foo': 1,
        'bar':
        {
            'x' : 10,
            'y' : 20,
            'z' : 30
        },
        'baz': 3
    },
    'c',
    'd',
    'e'
]
dt[2]['bar']['y']
```

```
[20]  ✓ 0.7s

...  20
```

Sets

Table of Contents

Definitions

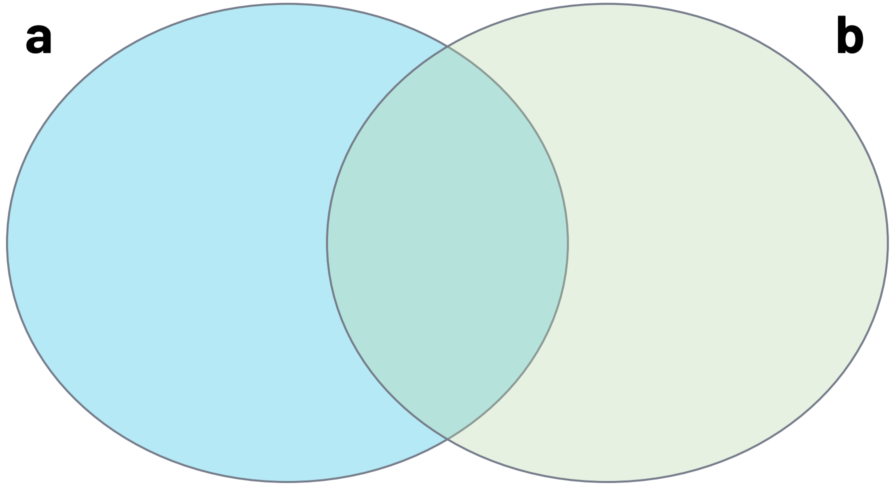


```
fruit = {'Apple', 'Orange', 'Banana'}
```

```
set()
```

Definitions

- No repetition
- Math operations
  - union
  - intersection
  - difference
- Unordered elements



Creating a `set`

Creating a `set`

- We have two basic ways to create a set.

- `{}`
- `set()`

```
set_1 = {'red', 'blue', 'pink', 'red'}
colors = 'red', 'blue', 'pink', 'red'
set_2 = set(colors)
print(set_1)
```

```
{'blue', 'pink', 'red'}
```

Creating a `set`

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

- `{}`
- `set()`

```
set_1 = {'red', 'blue', 'pink', 'red'}
colors = 'red', 'blue', 'pink', 'red'
set_2 = set(colors)
print(set_1)
print(set_2)
```

⚠ different order from the previous slide

```
{'red', 'blue', 'pink'}
{'red', 'blue', 'pink'}
```

Creating a `set`(review of pre-class)

- Here is an example of creating an empty `set`:

input :

```
empty_set = set()

print(type(empty_set))

```

output :

```
<class 'set'>

```

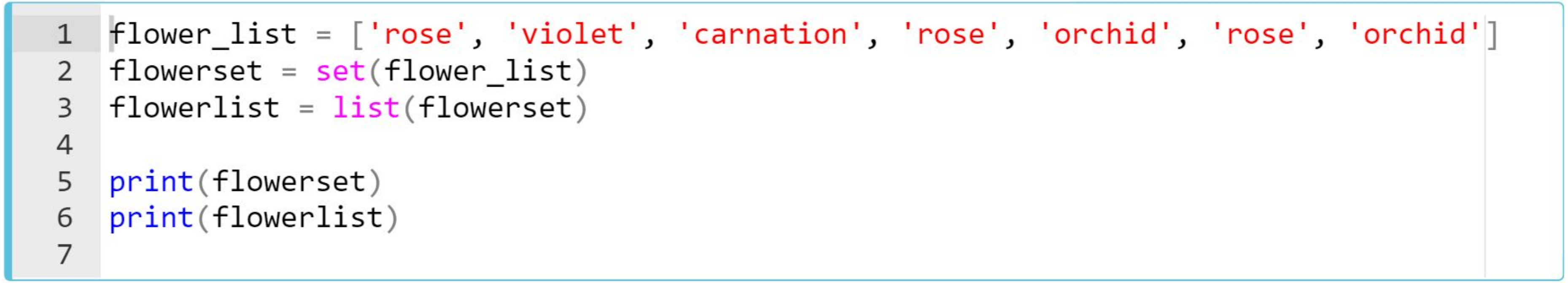
Creating a `set`

- Creating an empty set

~~`{}`~~

To create an empty `set`, you **can not** use 👉 `{}`. The only way to create an empty set is `set()` function.

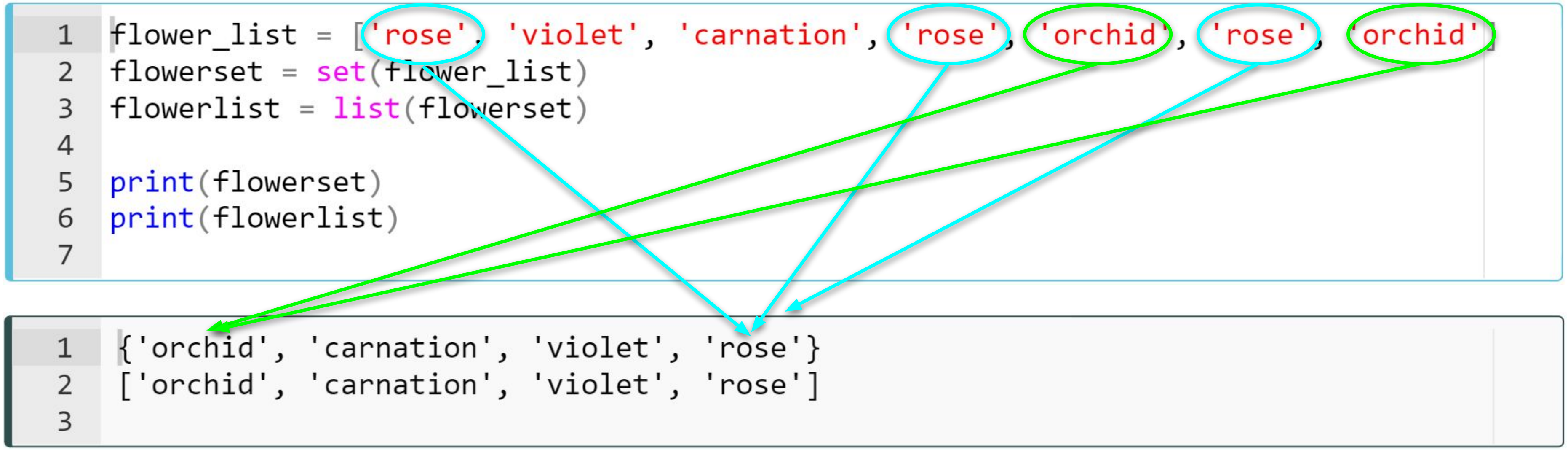
Creating a `set`(review of pre-class)



```python
flower_list = ['rose', 'violet', 'carnation', 'rose', 'orchid', 'rose', 'orchid']
flowerset = set(flower_list)
flowerlist = list(flowerset)

print(flowerset)
print(flowerlist)

```

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

Creating a set(review of pre-class)



```
flower_list = ['rose', 'violet', 'carnation', 'rose', 'orchid', 'rose', 'orchid']
flowerset = set(flower_list)
flowerlist = list(flowerset)

print(flowerset)
print(flowerlist)
```

```
{'orchid', 'carnation', 'violet', 'rose'}
['orchid', 'carnation', 'violet', 'rose']
```

Creating a `set`(review of pre-class)

- **Task** :
  - Do these two sets give the same output and why?

```
a = {'carnation', 'orchid', 'rose', 'violet'}
```

👇👆

```
b = {'rose', 'orchid', 'rose', 'violet', 'carnation'}
```

Creating a `set`

- The Answer is : **`True`**

```
{'carnation', 'orchid', 'rose', 'violet'}


{'rose', 'orchid', 'rose', 'violet', 'carnation'}
```

Main Operations with Sets

Main Operations with `sets` (review)

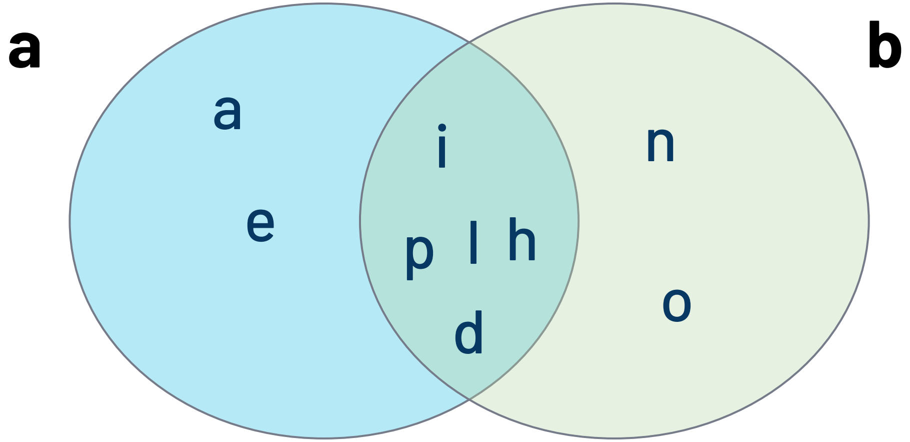
- The methods that can be used with `sets` :

- `.add()` : Adds a new item to the set.
- `.remove()` : Allows us to delete an item.
- `.intersection()` : Returns the intersection of two sets.
- `.union()` : Returns the unification of two sets.
- `.difference()` : Gets the difference of two sets.

Main Operations with `sets`

- Let's take a look these two sets below :

```python
a = set('philadelphia')
b = set('dolphin')
```

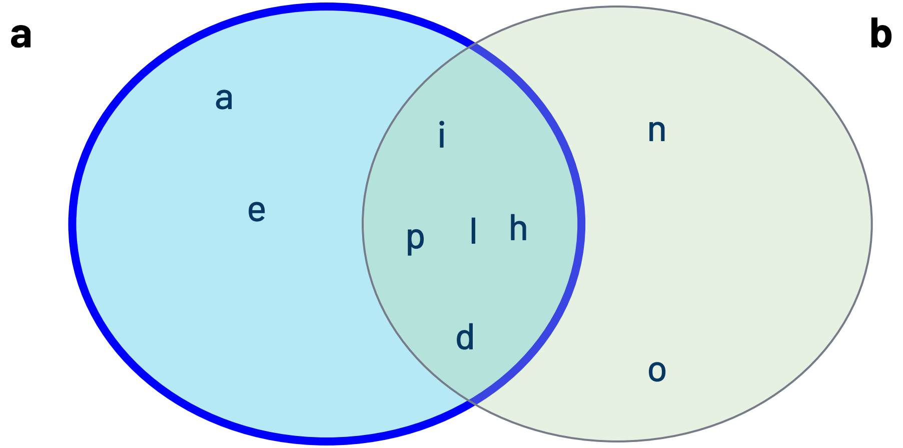


Main Operations with `sets`

- Let’s take a look these two sets below :

```
a = set('philadelphia')
print(a)
```

```
{'a', 'e', 'i', 'd', 'l', 'p', 'h'}
```

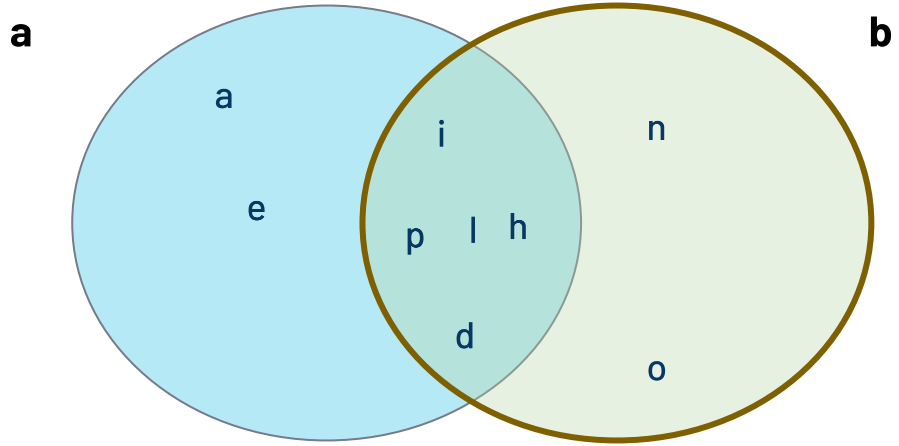


Main Operations with sets

- Let’s take a look these two sets below :

```
b = set('dolphin')
print(b)
```

```
{'d', 'l', 'o', 'p', 'n', 'i', 'h'}
```



CLARUSWAY©
WAY TO REINVENT YOURSELF

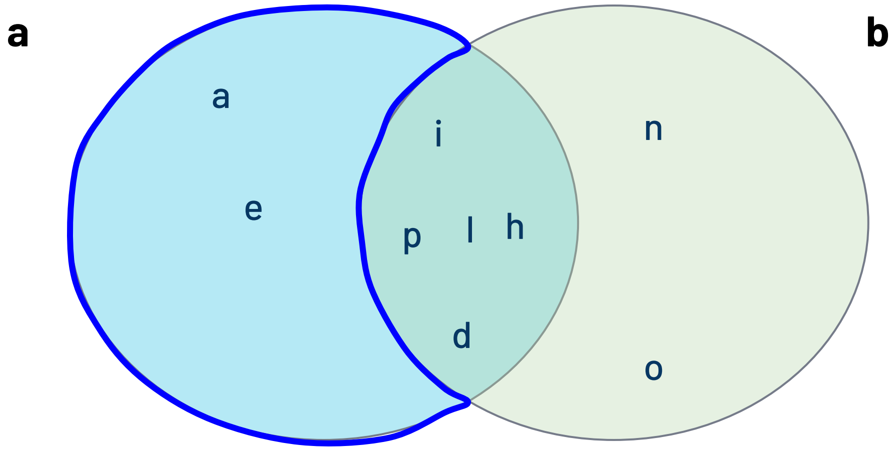
Main Operations with sets

- Basic set operations :

.difference(arg)

```
print(a - b)
print(a.difference(b))
```

```
{'a', 'e'}
```



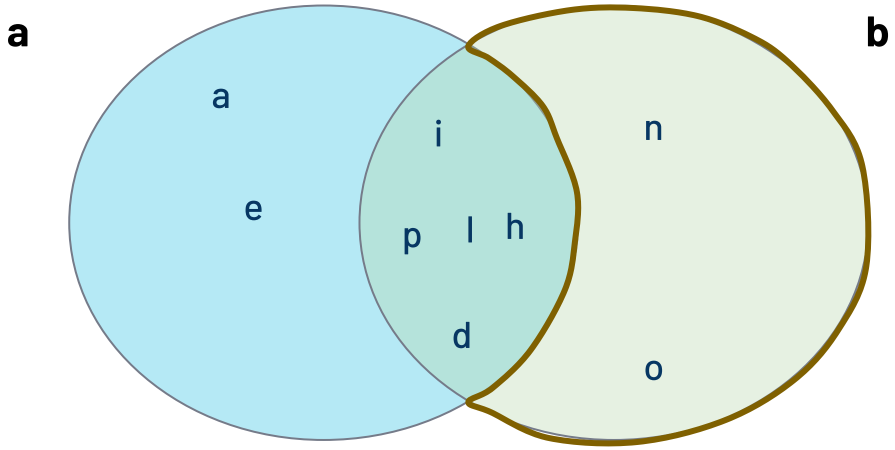
Main Operations with sets

- Basic set operations :

.difference(arg)

```
print(b - a)
print(b.difference(a))
```

```
{'n', 'o'}
```



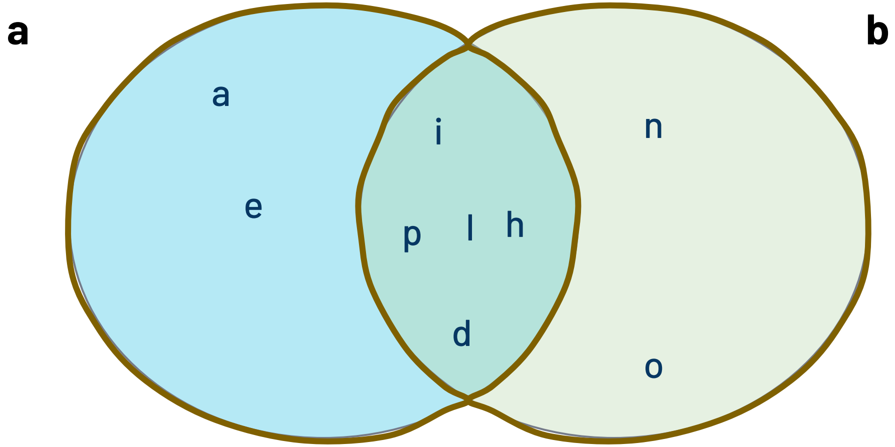
Main Operations with sets

- Basic set operations :

`.union(arg)`

```
print(a | b)
print(a.union(b))
```

```
{'p', 'h', 'i', 'l', 'd', 'o', 'n', 'a', 'e'}
```



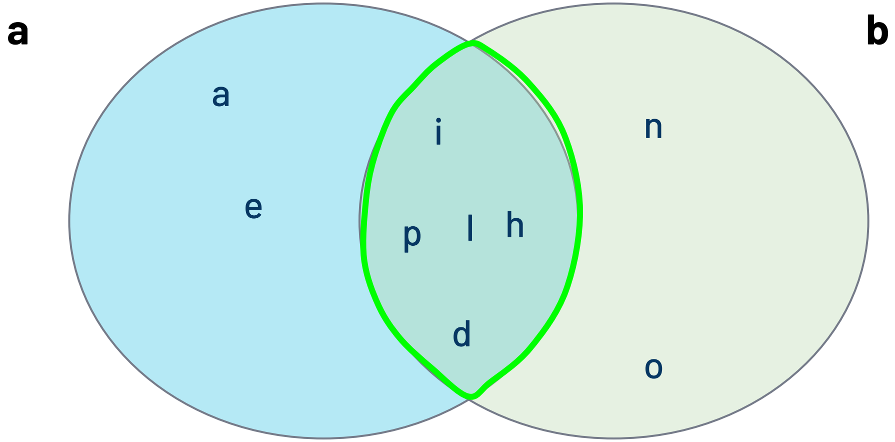
Main Operations with sets

- Basic set operations :

.intersection(arg)

```
print(a & b)
print(a.intersection(b))
```

```
{'p', 'h', 'i', 'l', 'd'}
```



Creating a `set`

- **Task** :
  - Let's create a `set` from which `str` type of the current date?
  - Date style would be "`mm/dd/yyyy`".
  - Creating a **set**, use both `set()` function and `{}` then figure out the results.

Creating a `set`

- The solution:

```python
a = set('09/01/2021')
b = {'09/01/2021'}
print(a)
print(b)
```

```
{'1', '0', '9', '2', '/'}
{'09/01/2021'}
```

Creating a `set`

- **Task** :

  Given a `list`, create a `set` to select and print the **unique** elements of the it.

```
given_list = [1, 2, 3, 3, 3, 3, 4, 4, 5, 5]
```

Creating a set

- **The code might be like** :

```
given_list = [1, 2, 3, 3, 3, 3, 4, 4, 5, 5]

unique = set(given_list)

print(unique)
```

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

Discuss in-class! Could you do the same thing using only curly braces `{}` instead of `set()` function?

Creating a `set`

- **Task** :

  -Create two `set`s of string data from the capitals of the **USA** and **New Zealand**. (**e.g**: `'Madrid' → convert into a set`)

  -Perform all set operations.

  - Intersection
  - Union
  - Difference

Creating a `set`

- **The code might be like** :

```
usa_capt = set('Washington')
nz_capt = set('Wellington')


print(usa_capt)
print(nz_capt)
```

```
{'h', 'W', 'a', 'o', 's', 'n', 'g', 'i', 't'}
{'W', 'o', 'l', 'e', 'n', 'g', 'i', 't'}
```

Creating a `set`

- **The code might be like** :

```python
usa_capt = set('Washington')
nz_capt = set('Wellington')

print(usa_capt - nz_capt)
print(usa_capt.difference(nz_capt))
```

```
{'s', 'h', 'a'}
{'s', 'h', 'a'}
```

Creating a `set`

- **The code might be like** :

```
usa_capt = set('Washington')
nz_capt = set('Wellington')

print(nz_capt - usa_capt)
print(nz_capt.difference(usa_capt))
```

```
{'l', 'e'}
{'l', 'e'}
```

Creating a `set`

- **The code might be like** :

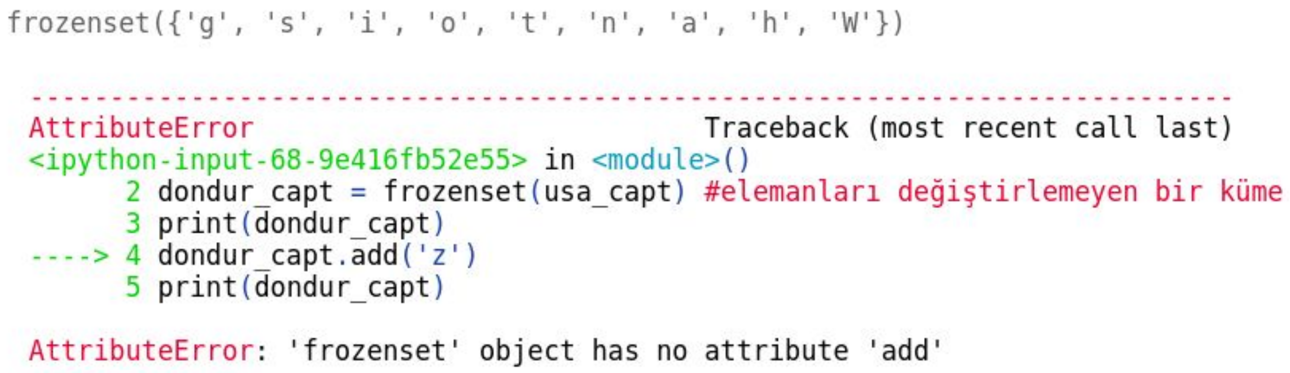
```
usa_capt = set('Washington')
nz_capt = set('Wellington')

print(nz_capt & usa_capt)
print(nz_capt.intersection(usa_capt))
```

```
{'i', 'o', 'g', 'n', 't', 'W'}
{'i', 'o', 'g', 'n', 't', 'W'}
```

frozenset()

- **Frozen set is just an immutable version of a Python set:**

```python
usa_capt = set('Washington')
dondur_capt = frozenset(usa_capt) #elements cannot be changed
print(dondur_capt)
dondur_capt.add('z')   ## ?
print(dondur_capt)
```



```
frozenset({'g', 's', 'i', 'o', 't', 'n', 'a', 'h', 'W'})

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-68-9e416fb52e55> in <module>()
      2 dondur_capt = frozenset(usa_capt) #elemanları değiştirlemeyen bir küme
      3 print(dondur_capt)
----> 4 dondur_capt.add('z')
      5 print(dondur_capt)

AttributeError: 'frozenset' object has no attribute 'add'
```

CLARUSWAY©
WAY TO REINVENT YOURSELF

String Methods

```
'capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'removeprefix', 'removesuffix', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill'
```

List

```
'append',
'clear',
'copy',
'count',
'extend',
'index',
'insert',
'pop',
'remove',
'reverse'
'sort'
```

Tuple

```
'count',
'index'
```

Dict

```
'clear',
'copy',
'fromkeys'
'get',
'items',
'keys',
'pop',
'popitem'
'setdefault
'update',
'values'
```

Set

```
'add', 'clear',
'copy',
'difference''diffe
rence_update',
'discard',
'intersection''int
ersection_update',
'isdisjoint',
'issubset',
'issuperset',
'pop', 'remove',
'symmetric_differe
nce',
'symmetric_differe
nce_update'union',
'update'
```

List	Tuple	Dict	Set
mutable	immutable	mutable	**mutable**
ordered	ordered	unordered	unordered
unhashable	hashable	unhashable	unhashable
iterable	hashable	no duplicate	iterable
duplicate	iterable		no duplicate
	duplicate		

Mutable or Immutable ?

List of immutable types:

```
int, float, decimal, complex, bool, string, tuple, range, frozenset, bytes
```

List of mutable types:

```
list, dict, set, bytearray, user-defined classes
```