1. SETS, TUPLES, DICTIONARIES

1A: TUPLES

A tuple consists of a number of values separated by commas

Creating tuple

```
# tuple with more than 1 value
new_tuple = 12, 15, 'hello', True
print(new_tuple) # (12, 15, 'hello')

# tuple with 0 values
empty_tuple = tuple()

#tuple with 1 value
singleton_tuple = 5,
print (singleton_tuple) # (5,)
```

Creating tuple also with list inside

```
new_tuple = 12,15, 'hello', True, [12, 15, False]
print(new_tuple) # (12, 15, 'hello', True, [12, 15, False])
```

Nested tuple

```
nested_tuple = (12,14), 16
print(nested_tuple ) # ((12, 14), 16)
```

Getting values

```
nested_tuple = (12,14), 16
print(nested_tuple[1]) # 16
print(nested_tuple[0][0]) # 12
```

Tuples are immutable (values cant be changed)

```
nested_tuple = 12,15, 'hello', True
nested_tuple[0] = 13 # TypeError: 'tuple' object does not support item
assignment
```

But elements could be mutable objects

```
nested_tuple = (12,14), 16, []
nested_tuple[2].append(3)
print(nested_tuple ) # ((12, 14), 16, [3])

nested_tuple = [12,15], 'hello', True
nested_tuple[0][1] = 13
print (nested_tuple) # ([12, 13], 'hello', True)
```

So …

```
list = [4, (5, 7), (2, 3)]

list[0] = 5 # okey
list[1] = 5 # okey
list[2][1] = 4 # TypeError: 'tuple' object does not support item assignment
```

Operators are working:

```
test_tuple = 'hello', 15, [14, 15, 'egg'], True

print(len(test_tuple)) # 4
print(15 in test_tuple) # True
print(test_tuple + (4,2)) # ('hello', 15, [14, 15, 'egg'], True, 4, 2)
print(test_tuple[2:4]) # ([14, 15, 'egg'], True)
print(test_tuple.index(15)) # 1
print(test_tuple.count('hello')) # 1
```

Converting list to tuple with list() function:

```
new_tuple = (16, 12, 15, True, 'hello')
new_list = list(new_tuple)

print(new_list) # [16, 12, 15, True, 'hello']
```

...and tuple() function:

```
a_list = [15, 14, [2, 3]]
a_string = 'I want to be a tuple'
print(tuple(a_list)) # (15, 14, [2, 3])
print(tuple(a_string)) # ('I', ' ', 'w', 'a', 'n', 't', ' ', 't', 'o', ' ',
'b', 'e', ' ', 'a', ' ', 't', 'u', 'p', 'l', 'e')
```

Iterating over tuple:

```
test_tuple = 'hello', 15, [14, 15, 'egg'], True
for i in test_tuple:
    print (i)
```

Values assigning “trick”:

```
normal_tuple = (12 ,16, 7)
x, y, z = normal_tuple
print (x) # 12
print (y) # 16
print (z) # 7
```

Why using tuples?

- immutable
- computationally cheaper to work with
- ?

1B: SETS

Mutable, unordered (!see below) collection of unique values

Creating a set:

```
# creating set with curly braces {}
new_set1 = {15, 17, False, 'tea'}

# empty set can be created with set() function
new_empty_set = set() # set()

# new set from string, list or tuple
new_set_string = set('creating a set') # {' ', 'i', 't', 'c', 'g', 'a', 'r',
'n', 's', 'e'}
new_set_list = set([12, 15]) # {12, 15}
new_set_tuple = set((15, True)) # {True, 15}
```

List cannot be inside set:

```
set_with_list = {15, 0, [2, 3]} # TypeError: unhashable type: 'list'
```

..but tuple is ok:

```
set_with_tuple = {15, 0, (2, 3)} # ok
```

Some operators are working:

```
test_set = {'element3', 15, True}

print(len(test_set)) # 3
print(15 in test_set) # True
```

Set has some methods:

```
# add() is alternative to append()
set_example = {15, 2, 0, 14}
set_example.add(7)
print(set_example) # {0, 7, 2, 14, 15}

# remove has problem when element is not in set
```

```
set_example = {15, 2, 0, 14}
set_example.remove(2)
print(set_example) # {0, 14, 15}
set_example.remove(158) # KeyError

# discard is without error when element is not presented
set_example = {15, 2, 0, 14}
set_example.discard(484564768)
print(set_example) # {0, 2, 14, 15}

# union
a_set = {5,1,6,7,3,2}
print(a_set.union({4,5})) # {1, 2, 3, 4, 5, 6, 7}

# intersection
a_set = {5,1,6,7,3,2}
print(a_set.intersection({1,4,5})) # {1, 5}

# difference
a_set = {5,1,6,7,3,2}
print(a_set.difference({1,4,5})) # {2, 3, 6, 7}
```

Iterating over set:

```
set_example = {15, 2, 0, 14}

for i in set_example:
    print (i)
```

Set is collection with unique values:

```
print(set('abrakadabra')) # {'b', 'a', 'r', 'k', 'd'}
print(set(((2,3), (2, 3)))) # {(2, 3)}
```

Why using sets?

- ordering

```
a_set = {5,1,6,7,3,2}
print(a_set) # {1, 2, 3, 5, 6, 7}
```

- membership testing
- removing duplicates
- mathematical operations - intersection, union, difference

1C: DICTIONARIES

Mutable collection of key value pairs

Creating dictionary:

```
a = dict(one=1, two=2, three=3)
b = {'one': 1, 'two': 2, 'three': 3}
c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
d = dict([('two', 2), ('one', 1), ('three', 3)])
e = dict({'three': 3, 'one': 1, 'two': 2})
```

The key is instead of index to access value (dictionaries are not ordered):

```
# so this is not working anymore
a_dict = {'a': 15, 'b': 17, 'c': 250}
a_dict[0] # KeyError: 0

# ...but this is okey now
a_dict['a'] # 15
```

Dictionaries are mutable:

```
a_dict = {'Bob': 45, 'Alice': 89, 'Cecilia': 250}
a_dict['Bob'] += 20
print(a_dict) # {'Alice': 89, 'Cecilia': 250, 'Bob': 65}
```

Values and keys can be strings/ numbers/ lists:

```
a_dict = {1: 'text_value', 'text_key': 89, 3: [80, False]}
```

Methods and operators of dictionary:

```
a_dict = {'a': 156, 'b': 89, 'c': 41, 'd': 547}
print(a_dict.items()) # dict_items([('b', 89), ('a', 156), ('d', 547), ('c',
41)])
print(a_dict.keys()) # dict_keys(['b', 'a', 'd', 'c'])
# list(a_dict.keys()) returns ['b', 'a', 'd', 'c']
print(a_dict.values()) # dict_values([89, 156, 547, 41])
# list(a_dict.values()) returns [89, 156, 547, 41]
print(len(a_dict)) # 4

# get() returns None if key is not defined
print(a_dict.get(4)) # None
print(a_dict.get('a')) # 156
```

Iterating over dictionary:

```
dict_example = {'a': 156, 'b': 89, 'c': 41, 'd': 547}

for i in dict_example:
    print (i) # d, b, a, c
```

```
for i in dict_example.values():
    print (i) # 547, 89, 156, 41

for i in dict_example.keys():
    print (i) # d, b, a, c
```

Why using dictionaries?

- switch-case (condition alternative):

```
time = int(input('what hour is it?')) # 10

what_to_do = {
    5: 'you should be sleeping',
    8: 'make a breakfast',
    10: 'have a coffee',
    15: 'go to shop',
    20: 'have a shower'
}

print(what_to_do[time]) # have a coffee
```

- making advanced constructions (db alternative):

```
countries_stats = {
    'Nigeria' : {
       'GDP': 1109000,
       'rank': 20,
       'languages': ['English']
    },
    'South Africa' : {
       'GDP': 725004,
       'rank': 30,
       'languages': ['Zulu', 'Xhosa', 'Afrikaans', 'English']
    },
    'Ethiopia' : {
       'GDP': 132000,
       'rank': 65,
       'languages': ['Amharic']
    }
}
print(countries_stats['Nigeria']['languages']) # ['English']
```

2. FUNCTIONS

"a block of code that could be callable"

used for:

- repeated code
- atomizing program

- making code more readable

examples from “real life”:

```
refrigerator = {
    "milk": 5,
    "eggs": 4,
    "cakes": 0,
}

# calling this function we add 3 eggs in refrigerator
def buy_eggs():
    print('adding 3 eggs to refrigerator')
    refrigerator["eggs"] += 3
    print('new eggs added to refrigerator')
    print('refrigerator', refrigerator)

# calling this function we add 1 milk in refrigerator
def buy_milk():
    print('adding 1 milk to refrigerator')
    refrigerator["milk"] += 1
    print('new milk added to refrigerator')
    print('refrigerator', refrigerator)

# calling this function we add 1 cake but use some eggs and milk
def make_cake():
    print("preparing cake")
    if (refrigerator["milk"] >= 2 and refrigerator["eggs"] >= 4):
        refrigerator["milk"] -= 2
        refrigerator["eggs"] -= 4
        refrigerator["cakes"] += 1
        print("we have a new cake in the refrigerator")
        print('refrigerator', refrigerator)
    else:
        print("sorry, we dont have enough of eggs or milk")

make_cake()

# preparing cake
# we have a new cake in the refrigerator
# refrigerator {'milk': 5, 'eggs': 4, 'cakes': 0}
```

note: function has to be defined before it is called

```
say_hello() # name 'say_hello' is not defined

def say_hello():
    print('hello there!')
```

2A: ARGUMENTS

arguments - values passed to function

```
# argument defines how many eggs are we adding
def buy_eggs(eggs):
    print('adding', eggs, 'eggs to refrigerator')
    refrigerator["eggs"] += eggs
    print('new eggs added to refrigerator')
    print('refrigerator', refrigerator)

# calling buy_eggs function with argument
buy_eggs(7)

# adding 7 eggs to refrigerator
# new eggs added to refrigerator
# refrigerator {'milk': 5, 'eggs': 11, 'cakes': 0}
```

setting default value as argument - in case of no argument passed, this value will be applied

```
# argument defines how many eggs are we adding
def buy_eggs(eggs = 3):
    print('adding', eggs, 'eggs to refrigerator')
    refrigerator["eggs"] += eggs
    print('new eggs added to refrigerator')
    print('refrigerator', refrigerator)


buy_eggs() # calling buy_eggs function without argument
# adding 3 eggs to refrigerator
# new eggs added to refrigerator
# refrigerator {'milk': 5, 'eggs': 7, 'cakes': 0}


buy_eggs(2) # calling buy_eggs function with argument
# adding 2 eggs to refrigerator
# new eggs added to refrigerator
# refrigerator {'milk': 5, 'eggs': 9, 'cakes': 0}
```

more arguments:

```
# defining how many eggs and milk do we need for a cake
def make_cake(eggs = 4, milk = 2):
    print("preparing cake")
    if (refrigerator["milk"] >= milk and refrigerator["eggs"] >= eggs):
        refrigerator["milk"] -= milk
        refrigerator["eggs"] -= eggs
        refrigerator["cakes"] += 1
```

```
            print("we have a new cake in the refrigerator")
            print('refrigerator', refrigerator)
        else:
            print("sorry, we dont have enough of eggs or milk")

# calling make_cake function, defining eggs to 3 and bottles of milks to 4
make_cake(3, 4)

# define number of eggs to 5, number of milk bottles will be default
make_cake(5)

# define number of milk bottles to 1, number of eggs will be default
make_cake(milk = 1)
```

*args keyword - used when we dont know how many arguments are we waiting for

```
def say_someting(name, weather, *args):
    print('hello, my name is', name)
    print('we have a', weather, 'weather')
    for a in args:
        print(a)

say_someting('Bob', 'sunny', 'carrot', 150, True)

#hello, my name is Bob
#we have a sunny weather
#carrot
#150
#True
```

2B: SCOPE - LOCAL VS GLOBAL VARIABLES

- variables defined inside the function are not defined outside of this function, ...
- global variables - variables defined outside of all functions, objects, …
    - could be read from anywhere
    - using “**global**” keyword when writing to global variable
- local variables could be used only inside the scope they were defined in

```
# example1 - another favourite_color is defined inside function
favourite_color = 'blue'

def change_favourite_color(new_color):
    favourite_color = new_color
    print('favourite_color should be changed to', favourite_color)

change_favourite_color('pink')
print(favourite_color) # blue
```

```
# example2 - reading global variable is ok everywhere
favourite_color = 'blue'

def what_is_my_favourite_color():
    print('my favourite color is', favourite_color) # blue

what_is_my_favourite_color()


# example3 - using keyword global to set value of global variable
favourite_color = 'blue'

def change_favourite_color(new_color):
    global favourite_color
    favourite_color = new_color

change_favourite_color('pink')
print(favourite_color) # finally pink


# example 4 - accessing local variable
def create_local_variable(local_value):
    local_variable = local_value
    print(local_variable) # 'hello local'

create_local_variable('hello local')
print(local_variable) # name 'local_variable' is not defined
```

2C: RETURN

- log of function in programming returns something as a output
- keyword “**return**” ends the function and sends a value back

```
def make_average(number_list):
    avg = sum(number_list)/len(number_list)
    return avg

print(make_average([3,2,4,6])) # 3.75


# making cakes example
refrigerator = {
    "milk": 5,
    "eggs": 4,
    "cakes": 0,
}
```

```python
# how many cakes we are able to make
def number_of_cakes(eggs = 3, milk = 1):
    max_eggs = refrigerator['eggs'] / eggs
    max_milk = refrigerator['milk'] / milk
    return min([max_eggs, max_milk])

print(number_of_cakes(1,1)) # 4 cakes could be made
print(number_of_cakes(4,1)) # only 1 cake could be made
```