1. What is the correct writing of the programming language that we used in this course?
( ) Phyton
( ) Pyhton
( ) Pthyon
( ) Python

2. What is the output of the code below?

```
my_name = "Bora Canbula"
print(my_name[2::-1])
```

( ) alu
( ) ula
( ) roB
( ) Bor

3. Which one is not a valid variable name?
( ) for_
( ) Manisa_Celal_Bayar_University
( ) IF
( ) not

4. What is the output of the code below?

```
for i in range(1, 5):
  print(f"{i:2d}{(i/2):4.2f}", end='')
```

```
( ) 010.50021.00031.50042.00
( )  10.50 21.00 31.50 42.00
( ) 1 0.5 2 1.0 3 1.5 4 2.0
( ) 100.5 201.0 301.5 402.0
```

5. Which one is the correct way to print Bora's age?

```
profs = [
  {"name": "Yener", "age": 25},
  {"name": "Bora", "age": 37},
  {"name": "Ali", "age": 42}
]
```

( ) profs["Bora"]["age"]
( ) profs[1][1]
( ) profs[1]["age"]

6. What is the output of the code below?

```
x = set([int(i/2) for i in range(8)])
print(x)
```

( ) {0, 1, 2, 3, 4, 5, 6, 7}
( ) {0, 1, 2, 3}
( ) {0, 0, 1, 1, 2, 2, 3, 3}
( ) {0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4}

7. What is the output of the code below?

```
x = set(i for i in range(0, 4, 2))
y = set(i for i in range(1, 5, 2))
print(x^y)
```

( ) {0, 1, 2, 3}
( ) {}
( ) {0, 8}
( ) SyntaxError: invalid syntax

8. Which of the following sequences is immutable?
( ) List
( ) Set
( ) Dictionary
( ) String

9. What is the output of the code below?

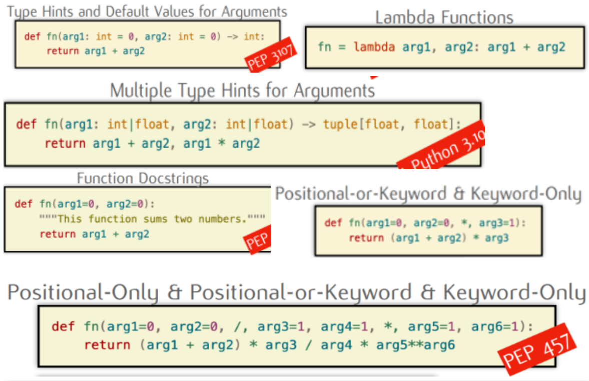
```
print(int(2_999_999.999))
```

( ) 2
( ) 3000000
( ) ValueError: invalid literal
( ) 2999999

10. What is the output of the code below

```
x = (1, 5, 1)
print(x, type(x))
```

( ) [1, 2, 3, 4] <class 'list'>
( ) (1, 5, 1) <class 'range'>
( ) (1, 5, 1) <class 'tuple'>
( ) (1, 2, 3, 4) <class 'set'>



Type Hints and Default Values for Arguments

```
def fn(arg1: int = 0, arg2: int = 0) -> int:
    return arg1 + arg2
```

PEP 3107

Lambda Functions

```
fn = lambda arg1, arg2: arg1 + arg2
```

Multiple Type Hints for Arguments

```
def fn(arg1: int|float, arg2: int|float) -> tuple[float, float]:
    return arg1 + arg2, arg1 * arg2
```

Python 3.10

Function Docstrings

```
def fn(arg1=0, arg2=0):
    """This function sums two numbers."""
    return arg1 + arg2
```

PEP

Positional-or-Keyword & Keyword-Only

```
def fn(arg1=0, arg2=0, *, arg3=1):
    return (arg1 + arg2) * arg3
```

Positional-Only & Positional-or-Keyword & Keyword-Only

```
def fn(arg1=0, arg2=0, /, arg3=1, arg4=1, *, arg5=1, arg6=1):
    return (arg1 + arg2) * arg3 / arg4 * arg5**arg6
```

PEP 457

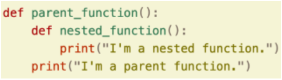
setattr(*object*, *name*, *value*)
This is the counterpart of getattr(). The arguments are an object, a string, and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, setattr(x, 'foobar', 123) is equivalent to x.foobar = 123.

name need not be a Python identifier as defined in Identifiers and keywords unless the object chooses to enforce that, for example in a custom __getattribute__() or via __slots__. An attribute whose name is not an identifier will not be accessible using the dot notation, but is accessible through getattr() etc..

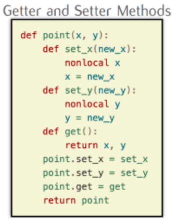
getattr(*object*, *name*)
getattr(*object*, *name*, *default*)
Return the value of the named attribute of *object*. *name* must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, getattr(x, 'foobar') is equivalent to x.foobar. If the named attribute does not exist, *default* is returned if provided, otherwise AttributeError is raised. *name* need not be a Python identifier (see setattr()).

hasattr(*object*, *name*)
The arguments are an object and a string. The result is True if the string is the name of one of the object's attributes, False if not. (This is implemented by calling getattr(object, name) and seeing whether it raises an AttributeError or not.)

delattr(*object*, *name*)
This is a relative of setattr(). The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, delattr(x, 'foobar') is equivalent to del x.foobar. *name* need not be a Python identifier (see setattr()).

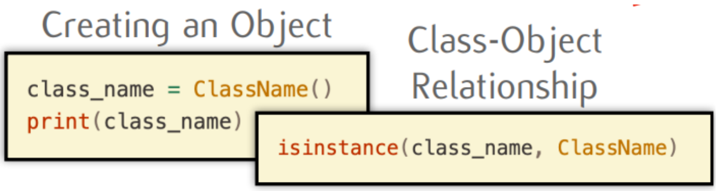


```
def parent_function():
    def nested_function():
        print("I'm a nested function.")
    print("I'm a parent function.")
```



Getter and Setter Methods

```
def point(x, y):
    def set_x(new_x):
        nonlocal x
        x = new_x
    def set_y(new_y):
        nonlocal y
        y = new_y
    def get():
        return x, y
    point.set_x = set_x
    point.set_y = set_y
    point.get = get
    return point
```

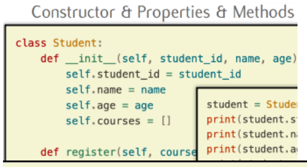


Creating an Object

```
class_name = ClassName()
print(class_name)
```

Class-Object Relationship

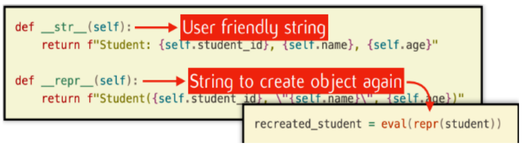
```
isinstance(class_name, ClassName)
```



Constructor & Properties & Methods

```
class Student:
    def __init__(self, student_id, name, age)
        self.student_id = student_id
        self.name = name
        self.age = age
        self.courses = []

    def register(self, course
```

```
student = Stude
print(student.s
print(student.n
print(student.a
```



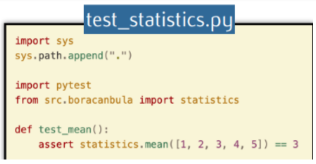
```
def __str__(self):
    return f"Student: {self.student_id}, {self.name}, {self.age}"
```

User friendly string

```
def __repr__(self):
    return f"Student({self.student_id}, \"{self.name}\", {self.age})"
```

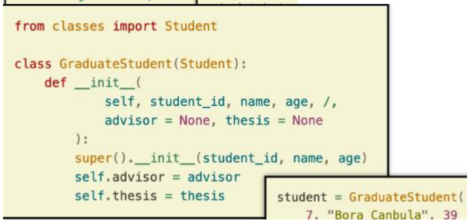
String to create object again

```
recreated_student = eval(repr(student))
```



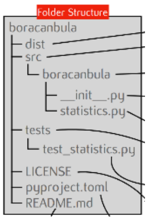
test_statistics.py

```
import sys
sys.path.append(".")

import pytest
from src.boracanbula import statistics

def test_mean():
    assert statistics.mean([1, 2, 3, 4, 5]) == 3
```

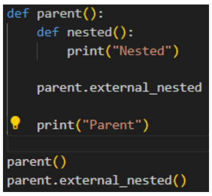


```
from classes import Student

class GraduateStudent(Student):
    def __init__(
            self, student_id, name, age, /,
            advisor = None, thesis = None
        ):
        super().__init__(student_id, name, age)
        self.advisor = advisor
        self.thesis = thesis
```

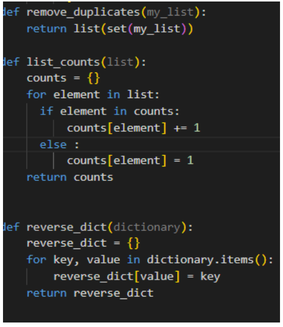
```
student = GraduateStudent(
    7, "Bora Canbula", 39
```



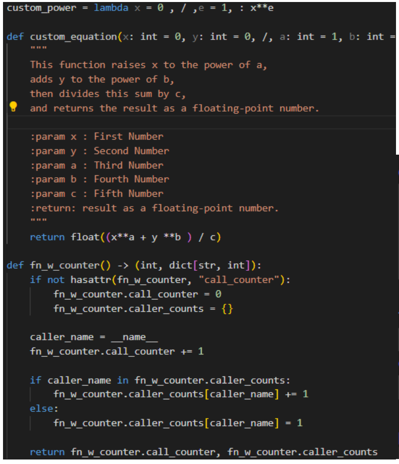
Folder Structure



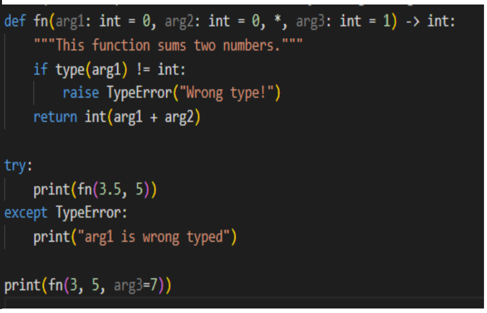
```
def parent():
    def nested():
        print("Nested")

    parent.external_nested

    print("Parent")

parent()
parent.external_nested()
```



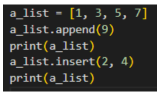
```
def remove_duplicates(my_list):
    return list(set(my_list))

def list_counts(list):
    counts = {}
    for element in list:
      if element in counts:
            counts[element] += 1
      else :
            counts[element] = 1
    return counts


def reverse_dict(dictionary):
    reverse_dict = {}
    for key, value in dictionary.items():
        reverse_dict[value] = key
    return reverse_dict
```



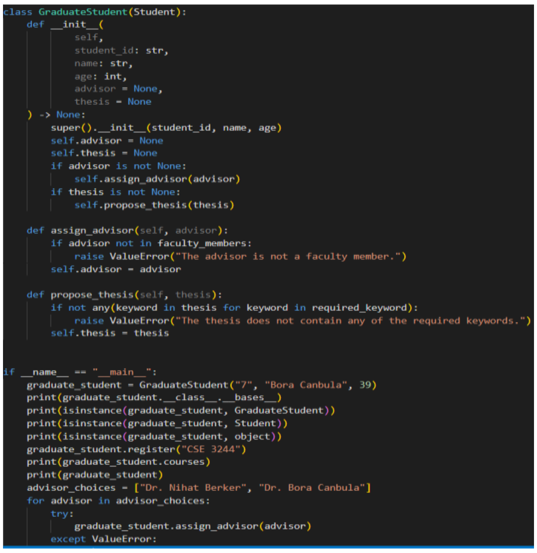
```
custom_power = lambda x = 0 , / ,e = 1, : x**e

def custom_equation(x: int = 0, y: int = 0, /, a: int = 1, b: int =
    """
    This function raises x to the power of a,
    adds y to the power of b,
    then divides this sum by c,
    and returns the result as a floating-point number.

    :param x : First Number
    :param y : Second Number
    :param a : Third Number
    :param b : Fourth Number
    :param c : Fifth Number
    :return: result as a floating-point number.
    """
    return float((x**a + y **b ) / c)

def fn_w_counter() -> (int, dict[str, int]):
    if not hasattr(fn_w_counter, "call_counter"):
        fn_w_counter.call_counter = 0
        fn_w_counter.caller_counts = {}

    caller_name = __name__
    fn_w_counter.call_counter += 1

    if caller_name in fn_w_counter.caller_counts:
        fn_w_counter.caller_counts[caller_name] += 1
    else:
        fn_w_counter.caller_counts[caller_name] = 1

    return fn_w_counter.call_counter, fn_w_counter.caller_counts
```



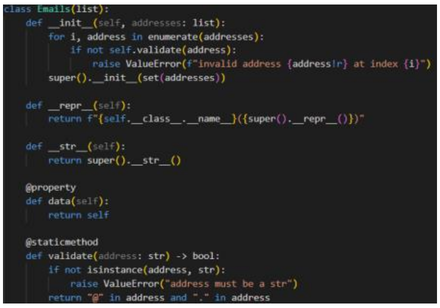
```
def fn(arg1: int = 0, arg2: int = 0, *, arg3: int = 1) -> int:
    """This function sums two numbers."""
    if type(arg1) != int:
        raise TypeError("Wrong type!")
    return int(arg1 + arg2)

try:
    print(fn(3.5, 5))
except TypeError:
    print("arg1 is wrong typed")

print(fn(3, 5, arg3=7))
```



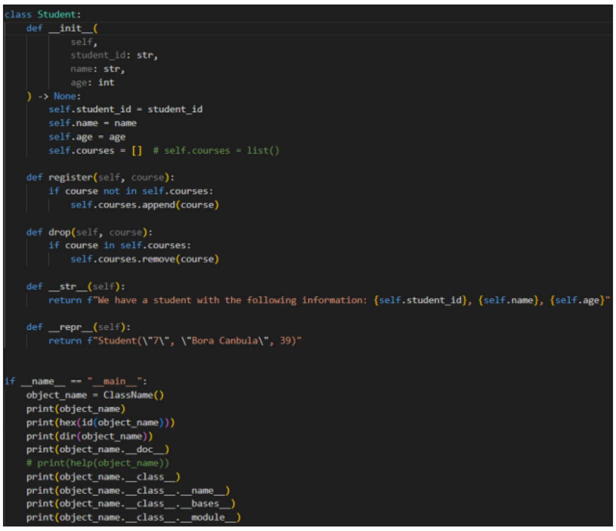
```python
a_list = [1, 3, 5, 7]
a_list.append(9)
print(a_list)
a_list.insert(2, 4)
print(a_list)
```



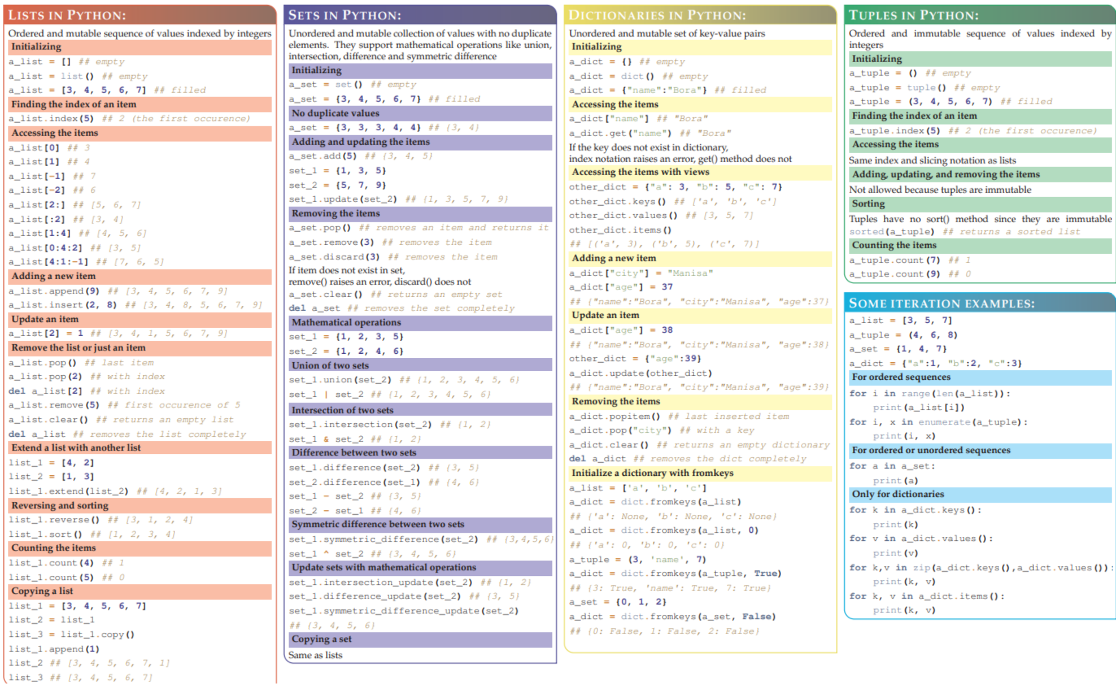
```python
class GraduateStudent(Student):
    def __init__(
            self,
            student_id: str,
            name: str,
            age: int,
            advisor = None,
            thesis = None
    ) -> None:
        super().__init__(student_id, name, age)
        self.advisor = None
        self.thesis = None
        if advisor is not None:
            self.assign_advisor(advisor)
        if thesis is not None:
            self.propose_thesis(thesis)

    def assign_advisor(self, advisor):
        if advisor not in faculty_members:
            raise ValueError("The advisor is not a faculty member.")
        self.advisor = advisor

    def propose_thesis(self, thesis):
        if not any(keyword in thesis for keyword in required_keyword):
            raise ValueError("The thesis does not contain any of the required keywords.")
        self.thesis = thesis


if __name__ == "__main__":
    graduate_student = GraduateStudent("7", "Bora Canbula", 39)
    print(graduate_student.__class__.__bases__)
    print(isinstance(graduate_student, GraduateStudent))
    print(isinstance(graduate_student, Student))
    print(isinstance(graduate_student, object))
    graduate_student.register("CSE 3244")
    print(graduate_student.courses)
    print(graduate_student)
    advisor_choices = ["Dr. Nihat Berker", "Dr. Bora Canbula"]
    for advisor in advisor_choices:
        try:
            graduate_student.assign_advisor(advisor)
        except ValueError:
```



```python
class Emails(list):
    def __init__(self, addresses: list):
        for i, address in enumerate(addresses):
            if not self.validate(address):
                raise ValueError(f"invalid address {address!r} at index {i}")
        super().__init__(set(addresses))

    def __repr__(self):
        return f"{self.__class__.__name__}({super().__repr__()})"

    def __str__(self):
        return super().__str__()

    @property
    def data(self):
        return self

    @staticmethod
    def validate(address: str) -> bool:
        if not isinstance(address, str):
            raise ValueError("address must be a str")
        return "@" in address and "." in address
```



```python
class Student:
    def __init__(
            self,
            student_id: str,
            name: str,
            age: int
    ) -> None:
        self.student_id = student_id
        self.name = name
        self.age = age
        self.courses = []  # self.courses = list()

    def register(self, course):
        if course not in self.courses:
            self.courses.append(course)

    def drop(self, course):
        if course in self.courses:
            self.courses.remove(course)

    def __str__(self):
        return f"We have a student with the following information: {self.student_id}, {self.name}, {self.age}"

    def __repr__(self):
        return f"Student(\"7\", \"Bora Canbula\", 39)"


if __name__ == "__main__":
    object_name = ClassName()
    print(object_name)
    print(hex(id(object_name)))
    print(dir(object_name))
    print(object_name.__doc__)
    # print(help(object_name))
    print(object_name.__class__)
    print(object_name.__class__.__name__)
    print(object_name.__class__.__bases__)
    print(object_name.__class__.__module__)
```



Lists in Python:

Ordered and mutable sequence of values indexed by integers

Initializing

```
a_list = [] ## empty
a_list = list() ## empty
a_list = [3, 4, 5, 6, 7] ## filled
```

Finding the index of an item

```
a_list.index(5) ## 2 (the first occurence)
```

Accessing the items

```
a_list[0] ## 3
a_list[1] ## 4
a_list[-1] ## 7
a_list[-2] ## 6
a_list[2:] ## [5, 6, 7]
a_list[:2] ## [3, 4]
a_list[1:4] ## [4, 5, 6]
a_list[0:4:2] ## [3, 5]
a_list[4:1:-1] ## [7, 6, 5]
```

Adding a new item

```
a_list.append(9) ## [3, 4, 5, 6, 7, 9]
a_list.insert(2, 8) ## [3, 4, 8, 5, 6, 7, 9]
```

Update an item

```
a_list[2] = 1 ## [3, 4, 1, 5, 6, 7, 9]
```

Remove the list or just an item

```
a_list.pop() ## last item
a_list.pop(2) ## with index
del a_list[2] ## with index
a_list.remove(5) ## first occurence of 5
a_list.clear() ## returns an empty list
del a_list ## removes the list completely
```

Extend a list with another list

```
list_1 = [4, 2]
list_2 = [1, 3]
list_1.extend(list_2) ## [4, 2, 1, 3]
```

Reversing and sorting

```
list_1.reverse() ## [3, 1, 2, 4]
list_1.sort() ## [1, 2, 3, 4]
```

Counting the items

```
list_1.count(4) ## 1
list_1.count(5) ## 0
```

Copying a list

```
list_1 = [3, 4, 5, 6, 7]
list_2 = list_1
list_3 = list_1.copy()
list_1.append(1)
list_2 ## [3, 4, 5, 6, 7, 1]
list_3 ## [3, 4, 5, 6, 7]
```

Sets in Python:

Unordered and mutable collection of values with no duplicate elements. They support mathematical operations like union, intersection, difference and symmetric difference

Initializing

```
a_set = set() ## empty
a_set = {3, 4, 5, 6, 7} ## filled
```

No duplicate values

```
a_set = {3, 3, 3, 4, 4} ## {3, 4}
```

Adding and updating the items

```
a_set.add(5) ## {3, 4, 5}
set_1 = {1, 3, 5}
set_2 = {5, 7, 9}
set_1.update(set_2) ## {1, 3, 5, 7, 9}
```

Removing the items

```
a_set.pop() ## removes an item and returns it
a_set.remove(3) ## removes the item
a_set.discard(3) ## removes the item
```

If item does not exist in set,
remove() raises an error, discard() does not

```
a_set.clear() ## returns an empty set
del a_set ## removes the set completely
```

Mathematical operations

```
set_1 = {1, 2, 3, 5}
set_2 = {1, 2, 4, 6}
```

Union of two sets

```
set_1.union(set_2) ## {1, 2, 3, 4, 5, 6}
set_1 | set_2 ## {1, 2, 3, 4, 5, 6}
```

Intersection of two sets

```
set_1.intersection(set_2) ## {1, 2}
set_1 & set_2 ## {1, 2}
```

Difference between two sets

```
set_1.difference(set_2) ## {3, 5}
set_2.difference(set_1) ## {4, 6}
set_1 - set_2 ## {3, 5}
set_2 - set_1 ## {4, 6}
```

Symmetric difference between two sets

```
set_1.symmetric_difference(set_2) ## {3,4,5,6}
set_1 ^ set_2 ## {3, 4, 5, 6}
```

Update sets with mathematical operations

```
set_1.intersection_update(set_2) ## {1, 2}
set_1.difference_update(set_2) ## {3, 5}
set_1.symmetric_difference_update(set_2)
## {3, 4, 5, 6}
```

Copying a set

Same as lists

Dictionaries in Python:

Unordered and mutable set of key-value pairs

Initializing

```
a_dict = {} ## empty
a_dict = dict() ## empty
a_dict = {"name":"Bora"} ## filled
```

Accessing the items

```
a_dict["name"] ## "Bora"
a_dict.get("name") ## "Bora"
```

If the key does not exist in dictionary,
index notation raises an error, get() method does not

Accessing the items with views

```
other_dict = {"a": 3, "b": 5, "c": 7}
other_dict.keys() ## ['a', 'b', 'c']
other_dict.values() ## [3, 5, 7]
other_dict.items()
## [('a', 3), ('b', 5), ('c', 7)]
```

Adding a new item

```
a_dict["city"] = "Manisa"
a_dict["age"] = 37
## {"name":"Bora", "city":"Manisa", "age":37}
```

Update an item

```
a_dict["age"] = 38
## {"name":"Bora", "city":"Manisa", "age":38}
other_dict = {"age":39}
a_dict.update(other_dict)
## {"name":"Bora", "city":"Manisa", "age":39}
```

Removing the items

```
a_dict.popitem() ## last inserted item
a_dict.pop("city") ## with a key
a_dict.clear() ## returns an empty dictionary
del a_dict ## removes the dict completely
```

Initialize a dictionary with fromkeys

```
a_list = ['a', 'b', 'c']
a_dict = dict.fromkeys(a_list)
## {'a': None, 'b': None, 'c': None}
a_dict = dict.fromkeys(a_list, 0)
## {'a': 0, 'b': 0, 'c': 0}
a_tuple = (3, 'name', 7)
a_dict = dict.fromkeys(a_tuple, True)
## {3: True, 'name': True, 7: True}
a_set = {0, 1, 2}
a_dict = dict.fromkeys(a_set, False)
## {0: False, 1: False, 2: False}
```

Tuples in Python:

Ordered and immutable sequence of values indexed by integers

Initializing

```
a_tuple = () ## empty
a_tuple = tuple() ## empty
a_tuple = (3, 4, 5, 6, 7) ## filled
```

Finding the index of an item

```
a_tuple.index(5) ## 2 (the first occurence)
```

Accessing the items

Same index and slicing notation as lists

Adding, updating, and removing the items

Not allowed because tuples are immutable

Sorting

Tuples have no sort() method since they are immutable

```
sorted(a_tuple) ## returns a sorted list
```

Counting the items

```
a_tuple.count(7) ## 1
a_tuple.count(9) ## 0
```

Some iteration examples:

```
a_list = [3, 5, 7]
a_tuple = (4, 6, 8)
a_set = {1, 4, 7}
a_dict = {"a":1, "b":2, "c":3}
```

For ordered sequences

```
for i in range(len(a_list)):
    print(a_list[i])
for i, x in enumerate(a_tuple):
    print(i, x)
```

For ordered or unordered sequences

```
for a in a_set:
    print(a)
```

Only for dictionaries

```
for k in a_dict.keys():
    print(k)
for v in a_dict.values():
    print(v)
for k,v in zip(a_dict.keys(),a_dict.values()):
    print(k, v)
for k, v in a_dict.items():
    print(k, v)
```