Final exam cheatsheet:

- import random
-
- Simple data structures:
  - List
  - Dictionaries
- Object oriented programming
  - Attribute declaration
- Algorithms:
  - Theory
  - Searching
  - Sorting
    - Bubble, selection and insertion examples
- Exams of other years:

Import random

random.randint([a,b])

Both ints a and b are included in this selection

Import classes

It is really important to import the class that`s being used in another file.

Simple data structures:

List

NOT for list inside list:

list = [1,2,[3,4]] list[1] = 2 list[2] = [3,4] list[2][1] = 4

Creation of a list inside another list: This example has a range of 2 and 3: OUTPUT: [[1,2,3][1,2,3]]

In [2]:

```
list = []

# Create List of list
for i in range(2):
    list.append([])
    for j in range(3):
        list[i].append(j)

print(list)
```

```
[[0, 1, 2], [0, 1, 2]]
```

Methods:

NOT: **list_name.method()**

- **append(element)** → adds an element at the end of the list. It is much more efficient to use **list_name.append(element)** than **list_name = list_name + [element]**
- **insert(index, element)**: inserts the element at the given position. If the position does not exist it will append it at the end. If you use a negative index, it will start counting from the end: **insert(-1, element)** is equivalent to append(element) -> **If inserting in a position, the element in that position goes to the right**
- l**ist1.extend(list2)**: appends all the elements of list2 to list1. Equivalent to **list1= list1 + list2** but using the + sign is less efficient as a copy of both lists is created and assigned to list1
- **index(element)**: returns the position of the element in the list. Error if the list does not contain the element (I should check first the element is in the list). If the element is repeated it returns the first position
- **index(element, start, end)**: returns the position of the element in the sublist between start and end (not included). end is optional.
- **count(element)**: returns the number of times the element is in list
- **clear()**: removes all the elements of the list. The list will be the empty one. With the **del(list)** function you also remove the variable, here the variable exists
- **remove(element)**: deletes the first appearance of the element in the list. If the element does not belong to the list, error. With **del(list([index])**, we remove giving the position, here we give the element
- **pop()**: removes and returns the last element. The element should be stored in another variable.
- **pop(index):** removes and returns the element at a given position. The element should be stored in another variable.
- **reverse()**: reverses the list
- **sort()**: sorts the list. It only works for lists made of similar elements (you cannot sort a list containing both string and numbers)

List functions instead of methods:

In [ ]:

```
# Compute the max value of a list: Returns the max value
def my_max(list: list):
    max_value = float(0)
    for i in range(len(list)):
        value = list[i]
        if value >= max_value:
            max_value = value
    return max_value

# Compute the  min value of a list: Returns the min value
def my_min(list: list):
    min_value = list[0]
    for i in range(len(list)):
        value = list[i]
        if value < min_value:
            min_value = value
    return min_value
```

Dictionaries

Methods

NOT	CODE	DEF
in → keys	print(”name” in person) → returns true	Bool function returns true when the value asked for is in the dicctionary
len(dic)	print(len(person))	Returns an int of the len of the dicctionary
. clear ()	print(person.clear())	
. get(Key)	print(person.get(”surname))	
. Keys()	keys_of_dic = list(person.keys())	prints the values
.values()	val = list(person.values())	prints the values of the dictionary(the value for name can be Paco), no order. IS A LIST
.items()	it = list(person.items())	prints touples, each touple contains the key and the value, no order. IS A LIST OF TOUPLES
.copy()	dict = person.compy()	Makes a shallow copy. We cant deep copy in dictionaries
.pop(key)	marks = person.pop(”marks”)	Takes the value of a key and saves it into a variables (marks in this case). Removes the key and the value from the dictionary
.update(dic)	person.update(dic2)	Updates a dictionary respect anohter dictionary. It adds the values of the dictionary in parenthesis to the first . NGD

Object Oriented programming:

Attribute declaration: (read only, private, etc...)

"Normal attributes" -> They can be changed and read from outside (have property and setter)

- When it is used as a parameter tu call the class

In [ ]:

```
class Test:
    # When the class Test is called, a parameter will be given for it
    def __init__(self, var_1: int):
        # The parameter is turned into a variable in the class
        self.var_1 = var_1

    # We create a property and a setter, the variable is now a parameter and can be read an
    @property
    def var_1(self):
        return self.__var_1

    @var_1.setter
    def var_1(self, var_1):
        # If the value given to the attribute is not of type int, we raise an error, if els
        if type(var_1) != int:
            raise TypeError("param_1 must be of type int")
        else:
            self.__var_1 = var_1
```

Read - only attributes: They can be read from outside but not changed. No declaration in innit.

- When it **can be computed from other parameters in the class or is constant**

In [1]:

```
# The property var_2 does not have a self.var_2 declaration. It is only declared as a prope
class Test:
    # When the class Test is called, a parameter will be given for it
    def __init__(self, var_1: int):
        # The parameter is turned into a variable in the class
        self.var_1 = var_1

    # We create a property and a setter, the variable is now a parameter and can be read an
    @property
    def var_1(self):
        return self.__var_1

    @var_1.setter
    def var_1(self, var_1):
        # If the value given to the attribute is not of type int, we raise an error, if els
        if type(var_1) != int:
            raise TypeError("param_1 must be of type int")
        else:
            self.__var_1 = var_1


    """ Changes here: The property var_2 is constantly the double of the var_1 property """
    @property
    def var_2(self):
        return self.var_1 * 2
```

Private attributes: They are only going to be used inside the class they are declared with two underscores before them.

- They can be given in the **innit** method as a parameter to give when declaring a class but then only used privately

In [ ]:

```
# The property var_3 is only declared in the innit method and is private, so only methods i
class Test:
    # When the class Test is called, a parameter will be given for it
    def __init__(self, var_1: int):
        # The parameter is turned into a variable in the class
        self.var_1 = var_1
        # In this case I could maybe use it as a constant
        self.__var_3 = 5

    # We create a property and a setter, the variable is now a parameter and can be read an
    @property
    def var_1(self):
        return self.__var_1

    @var_1.setter
    def var_1(self, var_1):
        # If the value given to the attribute is not of type int, we raise an error, if els
        if type(var_1) != int:
            raise TypeError("param_1 must be of type int")
        else:
            self.__var_1 = var_1


    """ Changes here: The property var_2 is constantly the double of the var_1 property """
    @property
    def var_2(self):
        return self.var_1 * 2
```

Magic methods:

All magic methods are written with two underscores before and after the name:

str(self)

In [ ]:

```
def __str__(self):
    """This method is invoked if I print the object. It must
    be called like this, no changes in name, no extra parameters.
    It must return the string I want to be shown when printing

    In order to return strings use: str() , "", + ."""
    return "someString"
```

repr(self)

In [ ]:

```
def __repr__(self):
    """This method is invoked if I write the name of the object in
    the interpreter. It must
    15
    be called like this, no changes in name, no extra parameters.
    It must return the string I want to be shown"""
    # I just invoke the str
    return self.__str__()
```

eq(self)

In [ ]:

```
def __eq__(self, another):
    """ This is the method Python will invoke if I try to compare
    two Date objects. In addition to self it must receive the other
    object. It must return true or false"""
    # Instead of creating a local variable we directly return the
    # result of the calculation
    # Two dates are equal if the values of all their attributes are
    # equal

    # This return is a BOOL
    return (self.var_1 == another.var_1
            and self.var_2 == another.var_2)
```

Type *Markdown* and LaTeX: α^2

Algorithms:

The big O notation represents the worst-case complexity of an algorithm. Defines the runtime required to execute an algorithm by identifying how the performance of the algorithm changes when the input size grows.

Searching

Linear search:

Go through the list and find the element you are looking for:

- list is not sorted
- complexity of O(n)

Binary search:

Go from the middle of the list towards the outer sides, choose one half of the list

- complexity is less: of O(log n) -> reduced by half
- The list has to be sorted in order to decide side

Sorting

Simple sorting methods: All of the ones used here have a complexity of O(n^2)

Bubble sort: (the slowest one)

Use a loop: Compare pairs of elements and change their position based on the sorting rule.

Algorithm for sorting with bubble sort:

In [1]:

```
# Sort from bigger to smaller
list_1 = [1,3,4,2,6,7,9]
for i in range(len(list_1)-1):
    for index in range(len(list_1)-2):
        if list_1[index] > list_1[index + 1]:
            list_1.insert(index + 1, list_1.pop(index))
print(list_1)
```

```
[1, 2, 3, 4, 6, 7, 9]
```

Selection sort: (fastest of the three)

Select the firs/last element of the desired list (given the rule), change that element to the first position. Do the same with every element.

Algorithm for sorting using selection sort:

In [12]:

```
# Sort from bigger to smaller -> Rule is that the next element is smaller than the last one
list_1 = [1,3,4,2,6,7,9]

# Iterate for every position that we are going to insert
for position_index in range(len(list_1)-1):
    # We start by assuming the position we are going to change has the max value. We are wo
    max_index = position_index

    # Check for every value in the interval from the position checking to the end of the li
    for check_index in range(position_index,len(list_1)):
        # If a position has a greater number, we take that position as the one to pop.
        if list_1[check_index] > list_1[max_index]: # A change in the equality here goes bi
            max_index = check_index
    # We pop the position with the greater number and insert it in the right place.
    list_1.insert(position_index,list_1.pop(max_index))

print(list_1)
```

```
[1, 2, 3, 4, 6, 7, 9]
```

Insertion sort

- The only one with external sorting -> It needs an external list We use another list, we add there the first element of our list. Then we keep adding elements, when we add an element, we compare it with the last element of the list and add it left or right of it. If we add it towards the side with more elements, we compare to all elements of that side until it sticks to a position

Algorithm for sorting using insertion sort:

In [ ]:

```
# Sort from bigger to smaller -> Rule is that the next element is smaller than the last one
list_1 = [1,3,4,2,6,7,9]
result = []
for insert_index in range(len(list_1)-1):
    result.insert(insert_index,list_1[insert_index])
    for index in range(len(list_1)-1):
```

Last year`s exams:

Creating a matrix and giving it values: Map matrix from January 19-20

- This works just try not to make rows or columns equal to 1. The minus one kind of makes all go haywire
- Because our matrix has a lot of zeroes, add an if statement declaring that any zero shouldn´t be taken into consideration

In [ ]:

```
# The second rule looks something like this:
rows = 2
columns = 2
list = []
for i in range(rows):
    for y in range(columns):
        # CHECK THAT THE VALUES ARE NOT 0
        if list[i][y] != 0:
            ...
```

In [3]:

```
   import random
list = []
name = "Wh0"
rows = 2
columns = 4
h_number = 2


def create_map_matrix(matrix, warehouse_name: str, rows: int, columns: int, house_number: i
    # Create a blank matrix with the right magnitude
    for x in range(rows):
        matrix.append([])
        for y in range(columns):
            matrix[x].append(0)

    # We first create a random position for the warehouse:
    warehouse_x = random.randint(0, rows-1)
    warehouse_y = random.randint(0, columns-1)
    matrix[warehouse_x][warehouse_y] = warehouse_name
    # Create random positions for the houses: If a position is already occupied look for an
    for i in range(house_number):
        random_x = random.randint(0, rows-1)
        random_y = random.randint(0, columns-1)
        while matrix[random_x][random_y] != 0:
            random_x = random.randint(1, rows-1)
            random_y = random.randint(1, columns-1)
        matrix[random_x][random_y] = f"h{i}"
    # fill the rest of the positions with streets
    for x in range(rows):
        for y in range(columns):
            if matrix[x][y] == 0:
                matrix[x][y] = "s"


create_map_matrix(list,name,rows,columns,h_number)
print(list)
```

```
[['s', 's', 'h1', 's'], ['s', 'Wh0', 'h0', 's']]
```

Careful, in order to add values at a given matrix: The matrix is going to be entirely used, first create the matrix: If else consider using the insertion of different list

In [ ]:

```
    """
    The streets creation matrix for the snowplowing:
    """

    def create_streets(self):
        """Create horizontal and vertical streets"""
        # create a list to then turn into a tuple
        streets = []
        # Create a blank matrix with the right magnitude
        for x in range(self.height):
            streets.append([])
            for y in range(self.width):
                streets[x].append(0)
        # create horizontal streets
        for i in range(self.height):
            random_lenght = random.randint(5, self.height)
            random_temperature = random.randint(-5, 4)
            streets[0][i] = Street("H", 0, i, random_temperature, random_lenght)
        # create vertical streets
        for i in range(self.width):
            random_lenght = random.randint(5, self.width)
            random_temperature = random.randint(-5, 4)
            streets[i][0] = Street("V", i, 0, random_temperature, random_lenght)
        # Set the street tuple
        self.streets = tuple(streets)
```