Dynamic Modelling Course - TEP4290: Warm-up 12

This exercise will help you to better understand how and why to comment and document, a skill that will be very important for your project work. Before completing the tasks below you should check these resources:

- https://www.youtube.com/watch?v=xuaTh5aor-Q "Real Python" 5 min video on commenting
- https://www.programiz.com/python-programming/docstrings Tutorial on docstrings
- https://pep8.org/ ultimate style guide for python
- Reasons to use type hints: https://www.youtube.com/watch?v=dgBCEB2jVU0
- type hints in python: https://peps.python.org/pep-0484/

You will practice some commenting and documenting in this exercise; the exercise is pass/fail.

Good luck!

Quick summary

You will have to comment and document the code you use in your project. While the computer executes your code, you, your partners, teachers, and anyone you ever ask for help have to **understand** your code from reading it.

Commenting

Fundamentally, a comment is a piece of text in your code that is not meant for execution, but rather for a human to read. In Jupyter Notebooks you can also use the markdown cells for this to some degree, but it is nice to differentiate between code specific comments and documenting your sources and approach in the markdown.

Guidelines for commenting

These can be up to taste:

From the Real Python tutorial:

- Keep comments as close as possible to the code that they describe.
- Don't use complex formatting such as tables or ASCII figures.
- Don't include redundant information.

- Design your code to comment itself.

From https://stackoverflow.blog/2021/12/23/best-practices-for-writing-code-comments/:

- Rule 1: Comments should not duplicate the code.
- Rule 2: Good comments do not excuse unclear code.
- Rule 3: If you can't write a clear comment, there may be a problem with the code.
- Rule 4: Comments should dispel confusion, not cause it.
- Rule 5: Explain unidiomatic code in comments.
- Rule 6: Provide links to the original source of copied code.
- Rule 7: Include links to external references where they will be most helpful.
- Rule 8: Add comments when fixing bugs.
- Rule 9: Use comments to mark incomplete implementations.

Documenting

You will not need an extensive documentation of your project, but it is good to understand the concept and practice a bit in this course. We do not require you to follow any specific syntax inside your docstrings, nor to use them for all functions or classes - but sometimes it can be quite helpful!

In [11]:
```
class ExampleClass:
    '''
    Makes room for documentation.

    Attributes:
    - example_string
    '''
    'some word'

    def __init__(self, some_string):
        '''Creates an instance and assigns some_string to example_string.'''
        self.example_string = some_string
        return

    def print(self):
        '''
        Prints and returns the example_string.

        Arguments:
        - none

        Returns:
        - example_string
        '''
        print(self.example_string)

        return self.example_string

print('printing the .__doc__ attribute:')
```

```
print(ExampleClass.__doc__)
print('printing the output of help(ExampleClass):')
print(help(ExampleClass))
print('printing the help(ExampleClass.print):')
print(help(ExampleClass.print))
```

```
printing the .__doc__ attribute:

    Makes room for documentation.

    Attributes:
    - example_string

printing the output of help(ExampleClass):
Help on class ExampleClass in module __main__:

class ExampleClass(builtins.object)
 |  ExampleClass(some_string)
 |
 |  Makes room for documentation.
 |
 |  Attributes:
 |  - example_string
 |
 |  Methods defined here:
 |
 |  __init__(self, some_string)
 |      Creates an instance and assigns some_string to example_string.
 |
 |  print(self)
 |      Prints and returns the example_string.
 |
 |      Arguments:
 |      - none
 |
 |      Returns:
 |      - example_string
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables
 |
 |  __weakref__
 |      list of weak references to the object

None
printing the help(ExampleClass.print):
Help on function print in module __main__:

print(self)
    Prints and returns the example_string.

    Arguments:
    - none

    Returns:
    - example_string

None
```

Type hints

In python you do not need to tell the type of a variable when you initialize it (unlike static typed languages like C++), but you can still declare a type in a type hint.
This might seem like extra work, but they make your code much more clear, so consider using them.
Consider the following function - you might not particularly care what the function does, but need to use it - so you want to insert a number. Without the type hint, you might be inclined to insert a float or an integer number, but the type hint tells you to only use the function with a string. You get further information about the function by the "-> bool", namely that the function only returns true or false.

In [12]:
```
def luhn_checksum(number:str)->bool:
    def digits_of(n:str):
        return [int(d) for d in n]
    digits:list = digits_of(number)
    odd_digits:list = digits[-1::-2]
    even_digits:list = digits[-2::-2]
    checksum:int = 0
    checksum += sum(odd_digits)
    for d in even_digits:
        checksum += sum(digits_of(d*2))
    return checksum % 10 == 0
```

Tasks

Below you find the complete code for a simple game of tic tac toe, split up in several functions. Your tasks will be to comment and document parts of the code. The last cell starts the game, so you can skip the tasks at first if you want to check what you are working for...

Source of the code: https://favtutor.com/blog-details/7-Python-Projects-For-Beginners

In [13]:
```
import random
```

Understand and comment

We start by making a board and some simple functions for it. Your task is to read the code given here, maybe do some tests in the cell below, and then **comment** the code (at least one comment per function).
Please also apply type hints!

In [14]:
```
#make a simple board with 10 empty spaces in a list
board:list = [' ' for x in range(10)]

def insertLetter(letter:str, pos:int):
```

```
    #change one cell (at pos) to the letter instead of empty
    board[pos] = letter

def spaceIsFree(pos:int) -> bool:
    #Check if a specific position on the board is free (empty).
    return board[pos] == ' '

def printBoard(board:list):
    print('   |   |   ')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
    print('   |   |   ')
    print('------------')
    print('   |   |   ')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
    print('   |   |   ')
    print('------------')
    print('   |   |   ')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
    print('   |   |   ')

def isBoardFull(board:list) -> bool:
    #check if the board is full and the game is tied
    if board.count(' ') > 1:
        return False
    else:
        return True

def IsWinner(b: list, l:str) -> bool:
    #check if the player has won the game
    return ((b[1] == l and b[2] == l and b[3] == l) or
    (b[4] == l and b[5] == l and b[6] == l) or
    (b[7] == l and b[8] == l and b[9] == l) or
    (b[1] == l and b[4] == l and b[7] == l) or
    (b[2] == l and b[5] == l and b[8] == l) or
    (b[3] == l and b[6] == l and b[9] == l) or
    (b[1] == l and b[5] == l and b[9] == l) or
    (b[3] == l and b[5] == l and b[7] == l))
```

In [15]:
```
#room for tests
len(board)
```

Out[15]: 10

Pseudocode

Pseudocode can be a good way to start thinking about code you want to write - instead of thinking about the syntax, you focus on *what* you want to do and write that in a comment.

Take the function playerMove and translate it into easily readable pseudocode.

In [16]:
```
def playerMove():
    run = True
    while run:
        move = input("please select a position to enter the X between 1 to 9")
```

```
        try:
            move = int(move)
            if move > 0 and move < 10:
                if spaceIsFree(move):
                    run = False
                    insertLetter('X' , move)
                else:
                    print('Sorry, this space is occupied')
            else:
                print('please type a number between 1 and 9')

        except:
            print('Please type a number')
```

In [17]:
```
#Your Pseudocode
#def playerMove():
    #while no input
        #ask for input
        #convert input to integer
        #if input is a number between 1 and 9
            #if space of the position is free
                #insert X at position
                #exit loop
            #else
                #print error message about space being occupied
        #else
            #print error message about input being out of range
    #if input is not a number
        #print message about input needing to be a number
```

Your code should be self explanatory

Mnemonic names can often make your code much more readable than comments - so take the function computerMove below and change the variable names such that the comments in it become obsolete.

Example: in line 3, change N to possible moves (and do the same wherever N occures), which makes the line "N = [x for x , letter in enumerate(board) if letter == ' ' and x != 0]" much more readable.

In [18]:
```
def computerMove():
    #create a list of all possible moves
    possible_moves = [x for x , letter in enumerate(board) if letter == ' ' and x !
    best_move = 0 # default best move is 0

    for y in ['O' , 'X']: #for both possible values of an occupied cell
        for i in possible_moves: #for all possible best_moves
            c = board[:] #copy board
            c[i] = y # set that cell to the current player

            if IsWinner(c, y): #does either player win there?
                best_move = i #that's the best move!
                return best_move
```

```
    cornersOpen = []
    for i in possible_moves:
        if i in [1 , 3 , 7 , 9]:
            cornersOpen.append(i)

    if len(cornersOpen) > 0:
        best_move = selectRandom(cornersOpen)
        return best_move

    if 5 in possible_moves:
        best_move = 5
        return best_move

    edgesOpen = []
    for i in possible_moves:
        if i in [2,4,6,8]:
            edgesOpen.append(i)

    if len(edgesOpen) > 0:
        best_move = selectRandom(edgesOpen)
        return best_move
```

Documentation of functions you know

Check the documentation for the print and display function.

In [19]:
```
help(print)
help(display)
```

```
Help on built-in function print in module builtins:

print(*args, sep=' ', end='\n', file=None, flush=False)
    Prints the values to a stream, or to sys.stdout by default.

    sep
      string inserted between values, default a space.
    end
      string appended after the last value, default a newline.
    file
      a file-like object (stream); defaults to the current sys.stdout.
    flush
      whether to forcibly flush the stream.

Help on function display in module IPython.core.display_functions:

display(*objs, include=None, exclude=None, metadata=None, transient=None, display_id
=None, raw=False, clear=False, **kwargs)
    Display a Python object in all frontends.

    By default all representations will be computed and sent to the frontends.
    Frontends can decide which representation is used and how.

    In terminal IPython this will be similar to using :func:`print`, for use in rich
er
    frontends see Jupyter notebook examples with rich display logic.

    Parameters
    ----------
    *objs : object
        The Python objects to display.
    raw : bool, optional
        Are the objects to be displayed already mimetype-keyed dicts of raw display
data,
        or Python objects that need to be formatted before display? [default: False]
    include : list, tuple or set, optional
        A list of format type strings (MIME types) to include in the
        format data dict. If this is set *only* the format types included
        in this list will be computed.
    exclude : list, tuple or set, optional
        A list of format type strings (MIME types) to exclude in the format
        data dict. If this is set all format types will be computed,
        except for those included in this argument.
    metadata : dict, optional
        A dictionary of metadata to associate with the output.
        mime-type keys in this dictionary will be associated with the individual
        representation formats, if they exist.
    transient : dict, optional
        A dictionary of transient data to associate with the output.
        Data in this dict should not be persisted to files (e.g. notebooks).
    display_id : str, bool optional
        Set an id for the display.
        This id can be used for updating this display area later via update_display.
        If given as `True`, generate a new `display_id`
    clear : bool, optional
        Should the output area be cleared before displaying anything? If True,
```

```
    this will wait for additional output before clearing. [default: False]
**kwargs : additional keyword-args, optional
    Additional keyword-arguments are passed through to the display publisher.

Returns
-------
handle: DisplayHandle
    Returns a handle on updatable displays for use with :func:`update_display`,
    if `display_id` is given. Returns :any:`None` if no `display_id` is given
    (default).

Examples
--------
>>> class Json(object):
...     def __init__(self, json):
...         self.json = json
...     def _repr_pretty_(self, pp, cycle):
...         import json
...         pp.text(json.dumps(self.json, indent=2))
...     def __repr__(self):
...         return str(self.json)
...

>>> d = Json({1:2, 3: {4:5}})

>>> print(d)
{1: 2, 3: {4: 5}}

>>> display(d)
{
  "1": 2,
  "3": {
    "4": 5
  }
}

>>> def int_formatter(integer, pp, cycle):
...     pp.text('I'*integer)

>>> plain = get_ipython().display_formatter.formatters['text/plain']
>>> plain.for_type(int, int_formatter)
<function _repr_pprint at 0x...>
>>> display(7-5)
II

>>> del plain.type_printers[int]
>>> display(7-5)
2

See Also
--------
:func:`update_display`

Notes
-----
In Python, objects can declare their textual representation using the
```

```
`__repr__` method. IPython expands on this idea and allows objects to declare
other, rich representations including:

  - HTML
  - JSON
  - PNG
  - JPEG
  - SVG
  - LaTeX

A single object can declare some or all of these representations; all are
handled by IPython's display system.

The main idea of the first approach is that you have to implement special
display methods when you define your class, one for each representation you
want to use. Here is a list of the names of the special methods and the
values they must return:

  - `_repr_html_`: return raw HTML as a string, or a tuple (see below).
  - `_repr_json_`: return a JSONable dict, or a tuple (see below).
  - `_repr_jpeg_`: return raw JPEG data, or a tuple (see below).
  - `_repr_png_`: return raw PNG data, or a tuple (see below).
  - `_repr_svg_`: return raw SVG data as a string, or a tuple (see below).
  - `_repr_latex_`: return LaTeX commands in a string surrounded by "$",
                    or a tuple (see below).
  - `_repr_mimebundle_`: return a full mimebundle containing the mapping
                         from all mimetypes to data.
                         Use this for any mime-type not listed above.

The above functions may also return the object's metadata alonside the
data.  If the metadata is available, the functions will return a tuple
containing the data and metadata, in that order.  If there is no metadata
available, then the functions will return the data only.

When you are directly writing your own classes, you can adapt them for
display in IPython by following the above approach. But in practice, you
often need to work with existing classes that you can't easily modify.

You can refer to the documentation on integrating with the display system in
order to register custom formatters for already existing types
(:ref:`integrating_rich_display`).

.. versionadded:: 5.4 display available without import
.. versionadded:: 6.1 display available without import

Since IPython 5.4 and 6.1 :func:`display` is automatically made available to
the user without import. If you are using display in a document that might
be used in a pure python context or with older version of IPython, use the
following import at the top of your file::

    from IPython.display import display
```

Document

Write a documention (as docstrings) for selectRandom and main, then print it.

In [20]:
```
def selectRandom(li):

    ln = len(li)
    r = random.randrange(0,ln)
    return li[r]

def main():
    """
    Main function to run the tic-tac-toe game.

    The function initializes the game, welcomes the user, and controls the game loo
    It alternates between the player's move and the computer's move until there is

    Steps:
    1. Display a welcome message and the initial empty board.
    2. Loop until the board is full:
       - Check if the player or computer has won.
       - Let the player make a move and update the board.
       - Let the computer make a move if the game is still ongoing.
    3. Display the game result (win, lose, or tie).
    """

    print("Welcome to the game!")
    printBoard(board)

    while not(isBoardFull(board)):
        if not(IsWinner(board , 'O')):
            playerMove()
            printBoard(board)
        else:
            print("sorry you loose!")
            break

        if not(IsWinner(board , 'X')) and not(isBoardFull(board)):
            move = computerMove()
            if move == 0:
                print(" ")
            else:
                insertLetter('O' , move)
                print('computer placed an o on position' , move , ':')
                printBoard(board)
        elif IsWinner(board , 'X'):
            print("you win!")
            break

    if isBoardFull(board):
        print("Tie game")
print(help(main))
```

```
Help on function main in module __main__:

main()
    Main function to run the tic-tac-toe game.

    The function initializes the game, welcomes the user, and controls the game loo
p.
    It alternates between the player's move and the computer's move until there is a
winner or a tie.

    Steps:
    1. Display a welcome message and the initial empty board.
    2. Loop until the board is full:
       - Check if the player or computer has won.
       - Let the player make a move and update the board.
       - Let the computer make a move if the game is still ongoing.
    3. Display the game result (win, lose, or tie).

None
```

Play the game

Now play some tic tac toe! just run the cell below to start.

In [21]:
```python
while True:
    x = input("Do you want to play again? (y/n)")
    if x.lower() == 'y':
        board = [' ' for x in range(10)]
        print('--------------------')
        main()
    else:
        break
```

```
--------------------
Welcome to the game!
   |   |
   |   |
   |   |
------------
   |   |
   |   |
   |   |
------------
   |   |
   |   |
   |   |
   |   |
 X |   |
   |   |
------------
   |   |
   |   |
   |   |
------------
   |   |
   |   |
   |   |
computer placed an o on position 9 :
   |   |
 X |   |
   |   |
------------
   |   |
   |   |
   |   |
------------
   |   |
   |   | O
   |   |
   |   |
 X | X |
   |   |
------------
   |   |
   |   |
   |   |
------------
   |   |
   |   | O
   |   |
computer placed an o on position 3 :
   |   |
 X | X | O
   |   |
------------
   |   |
   |   |
   |   |
------------
```

```
   |   |
   |   | O
   |   |
   |   |
 X | X | O
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
 X |   | O
   |   |
computer placed an o on position 6 :
   |   |
 X | X | O
   |   |
-----------
   |   |
   |   | O
   |   |
-----------
   |   |
 X |   | O
   |   |
sorry you loose!
```

Well Done!

Now use your skills in all future projects!