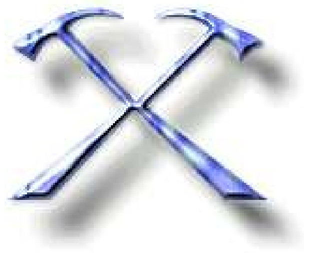
Lesson 2

Bjarne Poulsen



The Dictionary Data Type

The dictionary data type provides a flexible way to access and organize data.

Like a list, a dictionary is a collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. Indexes for dictionaries are called keys, and a key with its associated value is called a key-value pair.

In code, a dictionary is typed with braces, {}.

```
image = {'color': 'greyscale', 'size': 289983, 'type': 'jpg’,
             'address': 'https://upload.wikimedia.org/wikipedia/commons/7/7b/Moby_Dick_p510_illustration.jpg’}
image
```

```
{'address': 'https://upload.wikimedia.org/wikipedia/commons/7/7b/Moby_Dick_p510_illustration.jpg’,  'color': 'greyscale',
 'size': '289983’,  'type': 'jpg’}
```

This assigns a dictionary to the image variable. This dictionary’s keys are 'color', 'size', 'type', and 'address’.

The values for these keys are 'greyscale', 289983, 'jpg', and 'https://upload.wikimedia.org/wikipedia/commons/7/7b/Moby_Dick_p510_illustration.jpg', respectively.

You can access these values through their keys.

```
image['color']
```

```
'greyscale'
```

Adding New Key-Value Pairs

Dictionaries are dynamic structures, and you can add new key-value pairs to a dictionary at any time. For example, to add a new key-value pair, you would give the name of the dictionary followed by the new key in square brackets along with the new value.

```
image['source'] = 'Wikipedia’
image
```

```
{'address': 'https://upload.wikimedia.org/wikipedia/commons/7/7b/Moby_Dick_p510_illustration.jpg’, 'color': 'greyscale’, 'size': 289983, 'source': 'Wikipedia', 'type': 'jpg'}
```

Modifying Values in a Dictionary

To modify a value in a dictionary, give the name of the dictionary with the key in square brackets and then the new value you want associated with that key.

```
image['color'] = 'Black&White’
```

Removing Key-Value Pairs

When you no longer need a piece of information that’s stored in a dictionary, you can use the del statement to completely remove a key-value pair. All del needs is the name of the dictionary and the key that you want to remove.

```
del image['color']
```

The keys(), values(), and items() Methods

There are three dictionary methods that will return list-like values of the dictionary’s keys, values, or both keys and values: keys(), values(), and items(). The values returned by these methods are not true lists. They cannot be modified and do not have an append() method. But these data types (dict_keys, dict_values, and dict_items, respectively) can be used in for loops.

```
for key in image.keys():
    print(key)
```

Checking Whether a Key or Value Exists in a Dictionary

Recall from the previous session, that the in and not in operators can check whether a value exists in a list. You can also use these operators to see whether a certain key or value exists in a dictionary.

```
'color' in image.keys()
True
```

```
'compression' not in image.keys()
True
```

The get() Method

It is tedious to check whether a key exists in a dictionary before accessing that key’s value. Fortunately, dictionaries have a get() method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key does not exist.

```
image = {'color': 'greyscale', 'size': 289983, 'type': 'jpg',
               'address’:
'https://upload.wikimedia.org/wikipedia/commons/7/7b/Moby_Dick
_p510_illustration.jpg'}

color_val = image.get('color', 'unknown')
designer_val = image.get('designer', 'unknown')
designer_val
'unknown'
```

The setdefault() Method

You will often have to set a value in a dictionary for a certain key only if that key does not already have a value.

The setdefault() method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key does not exist. If the key does exist, the setdefault() method returns the key’s value.

```
# A simple character counter using the setdefault() method
fst_paragraph = '''Bjarne'''

count = {}
i=0
for character in fst_paragraph:
   count.setdefault(character, 0)
   count[character] += i

print(count)
```

```
{'B': 0, 'j': 0, 'a': 0, 'r': 0, 'n': 0, 'e': 0}
```

Nesting

Sometimes you will want to store a set of dictionaries in a list or a list of items as a value in a dictionary. This is called nesting. You can nest a set of dictionaries inside a list, a list of items inside a dictionary, or even a dictionary inside another dictionary. Nesting is a powerful feature, as the following examples will demonstrate.

In general, lists are useful to contain an ordered series of values, and dictionaries are useful for associating keys with values.

A List of Dictionaries

```
Aimage_0 = {'color': 'greyscale', 'size': 289983, 'type': 'jpg',
        'address':
'https://upload.wikimedia.org/wikipedia/commons/7/7b/Moby_Dick_p510_illustrat
ion.jpg’}

image_1 = {'color': 'greyscale', 'size': 492872, 'type': 'jpg',
        'address':
'https://upload.wikimedia.org/wikipedia/commons/f/f7/Queequeg.JPG’}

image_2 = {'color': 'greyscale', 'size': 497121, 'type': 'jpg',
        'address':
'https://upload.wikimedia.org/wikipedia/commons/8/8b/Moby_Dick_final_chase.j
pg'}

article_images = [image_0, image_1, image_2]
```

article_images List of Dictionaries

```
[{'address':
'https://upload.wikimedia.org/wikipedia/commons/7/7b/Moby_Dick_p51
0_illustration.jpg',
  'color': 'greyscale',
  'size': 289983,
  'type': 'jpg'},
 {'address':
'https://upload.wikimedia.org/wikipedia/commons/f/f7/Queequeg.JPG',
  'color': 'greyscale',
  'size': 492872,
  'type': 'jpg'},
 {'address':
'https://upload.wikimedia.org/wikipedia/commons/8/8b/Moby_Dick_fina
l_chase.jpg',
  'color': 'greyscale',
  'size': 497121,
  'type': 'jpg'}]
```

A List in a Dictionary

Rather than putting a dictionary inside a list, it is sometimes useful to put a list inside a dictionary.

```
images = {'color': 'greyscale', 'size': [289983, 492872, 497121], 'type': 'jpg',
        'address':
['https://upload.wikimedia.org/wikipedia/commons/7/7b/Moby_Dick_p510_illustration.jpg',

'https://upload.wikimedia.org/wikipedia/commons/f/f7/Queequeg.JPG',

'https://upload.wikimedia.org/wikipedia/commons/8/8b/Moby_Dick_final_chase.jpg']
        }

print(images['size'][2])
```

```
497121
```

```
for key, value in images.items():
    print("\n" + key.title())

    if type(value) == list:
        for element in value:
            print("\t * " + str(element))
    else:
        print("\t" + value)
```

Color

greyscale

Size

* 289983
* 492872
* 497121

Type

jpg

Address

* https://upload.wikimedia.org/wikipedia/commons/7/7b/Moby_Dick_p510_illustration.jpg
* https://upload.wikimedia.org/wikipedia/commons/f/f7/Queequeg.JPG
* https://upload.wikimedia.org/wikipedia/commons/8/8b/Moby_Dick_final_chase.jpg

A Dictionary in a Dictionary

You can nest a dictionary inside another dictionary, but your code can get complicated quickly when you do. For example, if you have several users for a website, each with a unique username, you can use the usernames as the keys in a dictionary. You can then store information about each user by using a dictionary as the value associated with their username.

In the following listing, we store three pieces of information about each user:

their first name, last name, and location.

We’ll access this information by looping through the usernames and the dictionary of information associated with each username:

```
users = {
   'aeinstein': {
      'first': 'albert',
      'last': 'einstein',
      'locations': ['princeton', 'copenhagen'],
      },

   'mcurie': {
         'first': 'marie',
         'last': 'curie',
         'locations': ['paris', 'athens'],
         },
}
```

```
for username, user_info in users.items():
    print("\nUsername: " + username)
    full_name = user_info['first'] + " " + user_info['last']
    locations = user_info['locations']

    print("\tFull name: " + full_name.title())
    for location in locations:
        print("\tLocation: " + location.title())
```

Username: aeinstein

- Full name: Albert Einstein
- Location: Princeton
- Location: Copenhagen

Username: mcurie

- Full name: Marie Curie
- Location: Paris
- Location: Athens

Object oriented programming in Python

Creating the Class.

The __init__() method is a special method Python runs automatically whenever we create a new instance based on the class. This method has two leading underscores and two trailing underscores, a convention that helps prevent Python's default method names from conflicting with your method names.

The self parameter is required in the method definition, and it must come first before the other parameters.

It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.

```
class Person:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def mamefunc(self):
    print("Hello my name is " + self.name)

p1 = Person("Bjarne", 60)
p1.mamefunc()
```

```
Hello my name is Bjarne
```

Delete Objects & Object Properties

```
del p1
```

```
del p1.age
```

Python Inheritance

When you create a child class, the parent class must be part of the current file and must appear before the child class in the file. The name of the parent class must be included in parentheses in the definition of the child class. The __init__() method takes the information required to make a instance.

The super() function is a special function that helps Python make connections between the parent and child class. It tells Python to call the __init__() method from the parent class, which gives a instance all the attributes of its parent class. The name super comes from a convention of calling the parent class a superclass and the child class a subclass.

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person):
  pass
```

Use the pass keyword when you do not want to add any other properties or methods to the class.

```
x = Student("Mike", "Olsen")
x.printname()
```

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width
```

```
# Here we declare that the Square class inherits from the
Rectangle class
class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)

    def perimeter(self):
        return 4 * self.length

square = Square(4)
print(square.area())

16
```

Python protected and private access modifiers

```
class Person:
  def __init__(self, name, age):
    self.__name = name
    self._age = age

  def mamefunc(self):
    print("Hello my name is " + self.__name)

class student(Person):
      pass
```

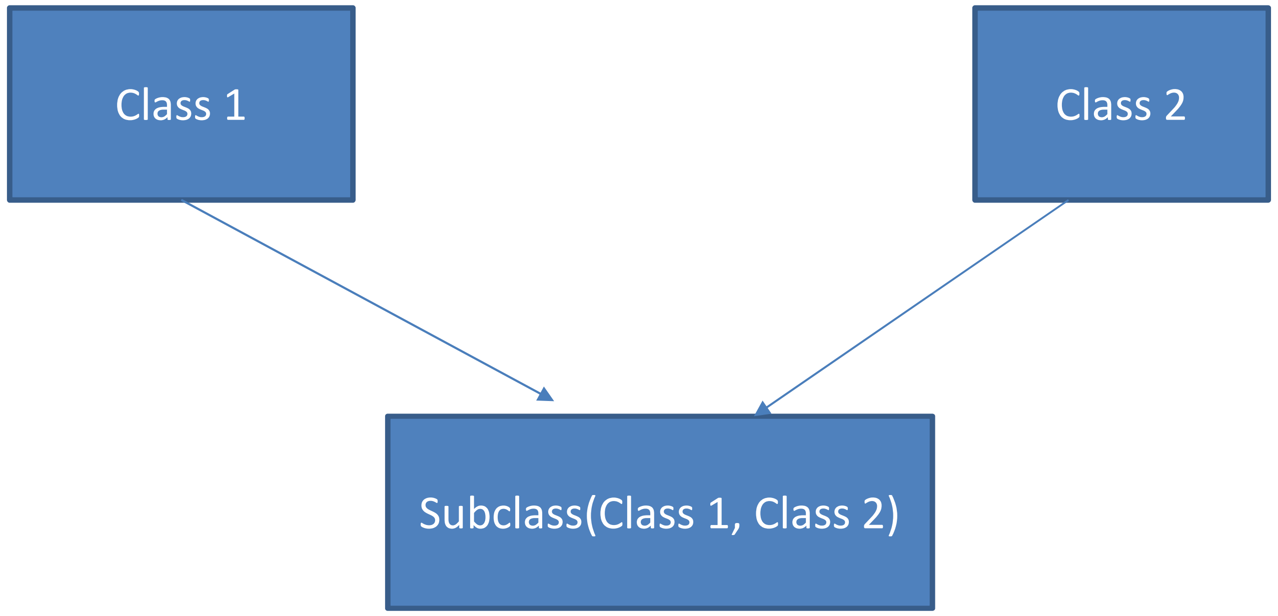
```
p1 = student("Bjarne", 60)
p1.mamefunc()

p2 = Person("Jens",42)
print(p2._age) ?
Print(p2.__name) ?
print(p2._Person__name) ?
```

```
class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)

class Cube(Square):
    def surface_area(self):
        face_area = super().area()
        return face_area * 6

    def volume(self):
        face_area = super().area()
        return face_area * self.length
```

super() returns a delegate object to a parent class, so you call the method you want directly on it: super().area().

Not only does this save us from having to rewrite the area calculations, but it also allows us to change the internal .area() logic in a single location. This is especially in handy when you have a number of subclasses inheriting from one superclass.

Multiple Inheritance Overview



```
class Person:
    #defining constructor
    def __init__(self, personName, personAge):
        self.name = personName
        self.age = personAge

    #defining class methods
    def showName(self):
        print(self.name)

    def showAge(self):
        print(self.age)
```

```
class Student:
  def __init__(self, studentId):
     self.studentId = studentId

  def getId(self):
     return self.studentId
```

```
# extends both Person and Student class
class Resident(Person, Student):
    def __init__(self, name, age, id):
        Person.__init__(self, name, age)
        Student.__init__(self, id)


# Create an object of the subclass
resident1 = Resident('John', 30, '102')
resident1.showName()
print(resident1.getId())
```

```
class A:
    def __init__(self):
        self.name = 'John'
        self.age = 23

    def getName(self):
        return self.name

class B:
    def __init__(self):
        self.name = 'Richard'
        self.id = '32'
    def getName(self):
        return self.name
```

```
class C(A, B):
    def __init__(self):
        A.__init__(self)
        B.__init__(self)

    def getName(self):
        return self.name


C1 = C()
print(C1.getName())
```

```
Richard
```

```
class C(A, B):
    def __init__(self):
    super().__init__()

    def getName(self):
        return self.name

C1 = C()
print(C1.getName())
```

```
John
```

Method Resolution Order

The method resolution order (or MRO) tells Python how to search for inherited methods. This comes in handy when you're using super() because the MRO tells you exactly where Python will look for a method you're calling with super() and in what order.

Every class has an .__mro__ attribute that allows us to inspect the order, so let's do that:

```
print(C.__mro__)
```

```
(<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class 'object'>)
```

https://realpython.com/python-super/

Polymorphism in Python ?

```
class Document:
    def __init__(self, name):
        self.name = name

    def show(self):
        raise NotImplementedError("Subclass must implement
abstract method")
```

```
class Pdf(Document):
    def show(self):
        return 'Show pdf contents!'

class Word(Document):
    def show(self):
        return 'Show word contents!'

documents = [Pdf('Document1’), Pdf('Document2’), Word('Document3')]

for document in documents:
    print (document.name + ': ' + document.show())
```

```
Document1: Show pdf contents!
Document2: Show pdf contents!
Document3: Show word contents!
```

Polymorphism with a Function

We can also create a function that can take any object, allowing for polymorphism.

```
class Document:
   def __init__(self, name):
      self.name = name


   def show(self):
      raise NotImplementedError("Subclass must implement
abstract method")
   def pr(self):
      raise NotImplementedError("Subclass must implement
abstract method")
```

```
class Pdf(Document):
    def show(self):
        return 'Show pdf contents!'
    def pr(self):
        print("Pdf")

class Word(Document):
    def show(self):
        return 'Show word contents!'
    def pr(self):
        print("Word")
```

```
def show(Document):
    Document.pr()

pdf1 = Pdf("Pdf")
W1 = Word("Word")

show(pdf1)
show(W1)
```

```
Pdf
Word
```

Advanced Basics

Function Annotations

(Function annotations are specified in PEP-3107.)

There are typically three aspects of a function that don’t deal with the code within it:

- a name,
- a set of arguments
- an optional docstring.

  (Custom docstring formats and adding custom attributes to the function object)

Sometimes, though, that is not quite enough to fully describe how the function works or how it should be used.

Here is a function foo() that takes three arguments:

def foo(a, b: 'annotating b', c: int) -> float: print(a + b + c)

Note that foo() returns nothing.

The first argument a is not annotated.

The second argument b is annotated with the string ‘annotating b’.

The third argument c is annotated with type int.

The return value is annotated with the type float.

Note the “->” syntax for annotating the return value.

foo("hej ", "Bjarne ","Poulsen") hej Bjarne Poulsen

foo(7, 8,9) 24

Accessing Function Annotations

All the annotations are stored in a dictionary called __annotations__, which itself is an attribute of the function.

```
print(foo.__annotations__)
```

```
{'b': 'annotating b', 'c': <class 'int'>, 'return': <class 'float'>}
```

```
def headline(text: str, align: bool = True) -> str:
    if align:
        return f"{text.title()}\n{'-' * len(text)}"
    else:
        return f" {text.title()} ".center(50, "o")

print(headline("python type checking"))
Print(headline("use mypy", align="center"))
```

```
Python Type Checking
----------------------------
```

Use Mypy

```
pip install mypy
```

- **mypy test.py**

```
test.py:11: error: Argument "align" to "headline" has
incompatible type "str"; expected "bool"
```

mypy

Mypy is an optional static type checker for Python that aims to combine the benefits of dynamic (or "duck") typing and static typing. Mypy combines the expressive power and convenience of Python with a powerful type system and compile-time type checking. Mypy type checks standard Python programs; run them using any Python VM with basically no runtime overhead.

http://mypy-lang.org/

```python
def headline(text: str, align: bool = True) -> str:
    if align:
        return f"{text.title()}\n{'-' * len(text)}"
    else:
        return f" {text.title()} ".center(50, "o")

print(headline("python type checking"))
print(headline("use mypy", align=False))
```

```
Python Type Checking
--------------------
oooooooooooooooooooo Use Mypy oooooooooooooooooooo
```

https://realpython.com/python-type-checking/

List Comprehensions

```
out = [] for value in range(15):
      if value > 10:
     out.append(value)
print(out)
print([value for value in range(15) if value > 10])
[11, 12, 13, 14]
[11, 12, 13, 14]

print(min([value for value in range(5) if value > 2]))
print({val: str(val) for val in range(11) if val > 6})
{7: '7', 8: '8', 9: '9', 10: '10’}

d1 =dict((val, str(val)) for val in range(11) if val > 6)
print(d1)
{7: '7', 8: '8', 9: '9', 10: '10'}
```

Functions creating iterators for efficient looping: itertools.

The itertools module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Together, they form an “iterator algebra” making it possible to construct specialized tools succinctly and efficiently in pure Python.

https://docs.python.org/3/library/itertools.html

```
import itertools
print(list(itertools.chain(range(2), range(4), range(6))))
```

```
[0, 1, 0, 1, 2, 3, 0, 1, 2, 3, 4, 5]
```

```
import itertools
print(list(zip(range(4), reversed(range(7)))))
```

```
[(0, 6), (1, 5), (2, 4), (3, 3)]
```

```
keys = map(chr, range(97, 102))
values = range(1, 6)
Print(dict(zip(keys, values)))
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
```

With Python 3, map() returns an iterator.

The map() function applies a given to function to each item of an iterable and returns a list of the results.

```
def calculateSquare(n):
  return n*n

numbers = (1, 2, 3)
result = map(calculateSquare, numbers)
print(result) ?

# converting map object to set
numbersSquare = set(result)
print(numbersSquare)
```

```
{1, 4, 9}
```

The Definition of Software Architecture

Software architecture is the process of converting software characteristics such as:

- flexibility
- scalability
- feasibility
- reusability
- security

A structured solution that meets the technical and the business expectations.

An architectural design pattern represents the practises, which experienced developers have gained through trial and error, when solving general problems (Sommerville, 2016).

Using an architectural design pattern will result in a strengthening of the general usability of the software as well of making the code less ambiguous. This will benefit the testing process, due to having a well structured code.

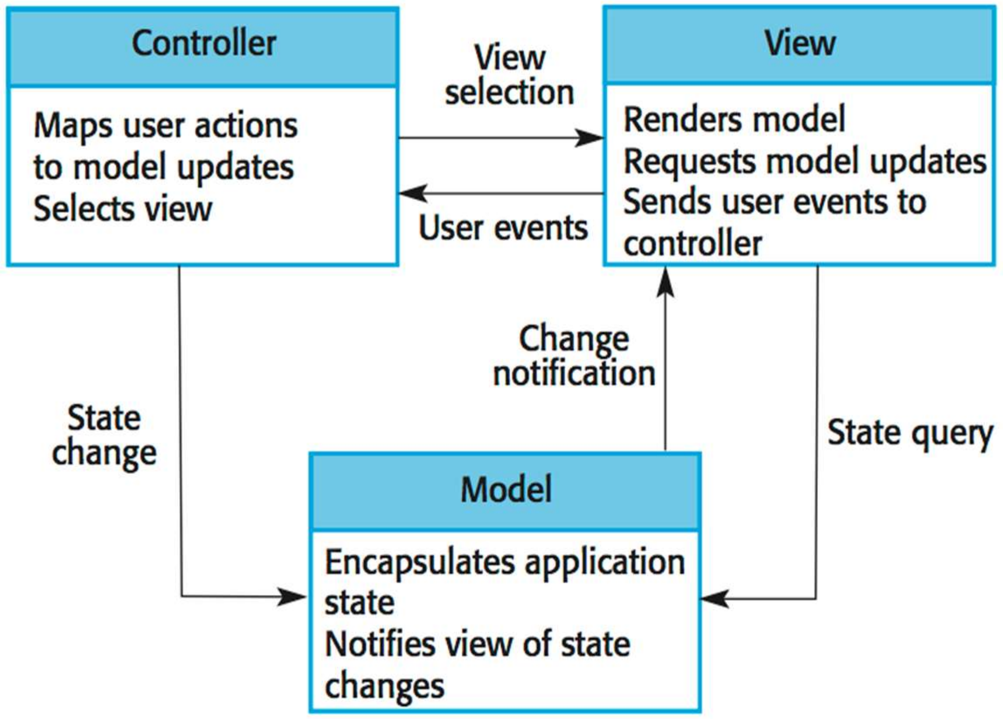
Software Architecture Patterns

An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context.

- Serverless Architecture.
- Event-Driven Architecture.
- Microservices Architecture.
- Master-slave pattern.
- ***Model-view-controller pattern***.

Model-view-controller pattern.

There is a a variety of available architectural patterns, suitable for different objects. The goal is to chose a pattern that can support the implementation of the system. Some patterns are standardized throughout practice in the development industry, and when choosing such a pattern, a general rule of thumb is to fit the pattern to the system. One of the most commonly used design patterns (Sommerville, 2016), is the Model-View-Controller (MVC) architecture.



State diagram showing the organization of the MVC design, into components, and the interaction between the components, as well as their internal logic. Figure from (Sommerville, 2016).

The ”**Model**” component, seen in the bottom of figure, manages the data and its representation, structure and behaviour along with the logic to update the View component whenever the data changes.

The ”**View**” component, seen in the top right of figure is the visual representation of the Model, and will manage and define how the data is presented. It is the component that provides the user with an output.

Lastly the ”**Controller**” component, seen in the top left of figure is the link between user interaction and the two other components. This component allows the user to manipulate the data contained in the Model. The received input can result in a change in data, invoking of methods in the Model or change the visual output produced by the View. These actions seldom occur in isolation.

Some general requirements which are essential for a MVC design can be extracted from these descriptions of the components:

1. All data storage should be in the Model component
2. All user interactivity should go through the Controller component
3. The user should not be able to directly interact with the Model component
4. All visual representations of the data should be created by the View component

A simple example in Python.

https://realpython.com/the-model-view-controller-mvc-paradigm-summarized-with-legos/

https://www.giacomodebidda.com/mvc-pattern-in-python-introduction-and-basicmodel/

Exercise for your project:

1. Analyze MVC so that you get a good understanding of how you will use it.
2. Design the first MVC prototype for your project.
3. Implements your design in Python.

4. Make a proof of concept. Which is:

Proof of Concept is a general approach that involves testing a certain assumption in order to obtain confirmation that the idea is feasible, viable and applicable in practice. In other words, it shows whether the software product or its separate function is suitable for solving a particular business problem.

5. Prepare a page with your results, as well as your code as an appendix. Review it with your review group and then get it approved by me. You have 2 weeks.