Hour 1

Review Classes and Objects:

- Creating a class is defining your own type.
- Combine data and functions into one package
- Everything in python is some type of object (ex: floats, ints, lists, strings etc)
- Classes are blueprint to make objects
- Objects are mutable

Review Attributes:

- Objects track their own data called attributes or instance variables
- In other programming languages, these can be defined as private or public.
- Everything in python is public by default.

Designing Employee class:

- Constructor:

```
class Employee:
    def __init__(self, id_num, name, position="Employee", salary=0):
        self.id = id_num
        self.name = name
        self.salary = salary
        self.position = position
```

- Getters

```
# Getters:
def get_name(self):
    return self.name

def get_id(self):
    return self.id

def get_salary(self):
    return self.salary

def get_position(self):
    return self.position
```

- Setters and methods:

```
# Setters
def set_salary(self, salary):
    self.salary = salary

def set_position(self, position):
    self.position = position

# Methods
def calc_pay(self, hours):
    return f"{hours * self.salary}$"

def __eq__(self, value):
    return self.id == value.id
```

```
def __str__(self):
    return f"{self.name} - {self.position}"
```

- Testing using unit test:

```
class TestEmployee(unittest.TestCase):
    def test_init(self):
        emp = Employee(1, "Chandler Bing", "Friend", 10)
        self.assertEqual(emp.name, "Chandler Bing")

        self.assertEqual(emp.id, 1)
        self.assertEqual(emp.salary, 10)

    def test_pay(self):
        emp = Employee(2, "Shiva", "God", 15)
        self.assertEqual(emp.calc_pay(30), 450)

    def test_bad_pay(self):
        with self.assertRaises(ValueError):
            emp = Employee(2, "Doug Judy", "Pontiac Bandit", 30)
            self.assertEqual(emp.calc_pay(-5), 0)
```

Hour 2

Protocols and Dynamic Typing:

- Also called interfaces in other languages
- Set of related methods
- Python is dynamically typed

Abstract Datatypes (ADTs):

- Data types who's implementation is hidden from user
- Generally implements some protocols but doesn't allow to see how its implemented.
- Examples: Stacks and Queues

DATA STRUCTURES

Stacks:

- LIFO - Last in First out
- Like an actual stack, stack of books.
- Simple and efficient
- In stacks, when you push item, it is added to the top of the list (end).
- When you pop, just the last item is removed.
- All of this happens in constant time and so is efficient
- A stack needs to be able to perform these operations:

Stack Operations

- push(): Put a new item on the stack
- pop(): Get the top item from the stack, removing that item
- peek() / top(): Look to see what the top element is
- create() / destroy(): Create or dispose of a stack
- is_empty(): Answers if the stack is empty or not (True/False)

Implementation:

```
class Stack:
    def __init__(self, lst=[]):
        self.stack = lst

    def push(self, element):
        self.stack.append(element)

    def pop(self):
        self.stack.pop()

    def peek(self):
        return self.stack[-1] if self.stack else None

    def is_empty(self):
        return not self.stack
```

Binary to Decimal using stacks:

```
def binary_to_decimal(binary):

    # Need to make a stack
    stack = Stack()

    # Takes in binary number like 110101101
    for num in binary:
        stack.push(num)

    # Take the top most number, multiply by 2 ^ i (i from 0 till stack is empty)
    i = 0
    decimal = 0
    while not stack.is_empty():
        print(stack.stack)
        # Keep adding top to a sum thing
        decimal += int(stack.pop()) * (2 ** i)

        # Increase index num
        i += 1


    return decimal
```

Hour 3

DATA STRUCTURE

Queue:

- First in first out. FIFO
- Like a line at an airport (First person in line gets served first)

Chaining Functions :

- This should be put in a separate file and each class should have a separate file of its own.

Predicates:

- Predicates (functions) take in a single value and return a True or False value.
- Checks if a statement is true or false

```
def less_than_ten(x): # predicate returns true or false
    return x < 10
```

Functions are also objects that can be stored and called:

```
import random

def abbra():
    print("No tricks up my sleeves")

def kadabara():
    print("No Magic hats")

def bunny():
    print("Viola! a  bunny rabbit")

FUNCTIONS = [abbra, kadabara, bunny]

def main():
    a = FUNCTIONS[1]
    a()
    b = random.choice(FUNCTIONS)
    b()
    b = random.choice(FUNCTIONS)
    b()

if __name__ == "__main__":
    main()
```

Hour 3. 5

Lambda:

- Single use callable function that is only used once
- `X = lambda a: a + 1`
- This would assign x as a + 1 for whatever a is.
- That's all it does and it is disposed off after that.

Review Nested Function:

```
1 ### Nested function example
2
3 def bouncer(password):
4     def inner():
5         print("You are now listening to my inner thoughts")
6         print("It's very quiet!")
7
8     def inner_v2():
9         print("yay")
10
11    if password.lower() == "speakeasy":
12        return inner
13    elif password.lower() == "mark":
14        return inner_v2
15    else:
16        print("Sorry Bub, the Bouncer says you must leave")
17
18
19 def main():
20    function = bouncer("open sesame")
21    if function is not None:
22        function()
23
24    function = bouncer("mark")
25    if not function is None:
26        function()
```