Oblig 4

Oppgave 1.1:

Exeption er en måte å håndtere feil i koden på, som unngår at koden kræsjer, og som lar den fortsette å kjøre. Som for eksempel:

```
try:
        result = 10 / 0
except ZeroDivisionError:
        print("Division by zero is not allowed.")
else:
        print("Division was successful. The result is:", result)
```

Vil prøve å dele 10 på null, hvis det ikke funker, vil den printe “Division by zero is not allowed.”, ellers, hvis koden kjører som normalt, vil den printe “Division was succelssful. The result is: {resultat}”.

Oppgave 1.2:

En klasse er en konstruktør for objekter i Python. Eksempel:

```
class Klasse:
    x = 10
objekt= Klasse()
print(objekt.x)
```

Vil da konstruere et objekt(“objekt”) som har en verdi, “objekt.x”, som er 10

En annen måte å konstruere objekter fra en klasse er som så:

```
class Klasse:
    def __init__(self, x):
        self.x = x
objekt= Klasse(5)
objekt2= Klasse(10)
print(objekt.x)
print(objekt2.x)
```

Her kan man se at de to objektene får to forskjellige verdier, selv om de er konstruert fra samme klasse. Man kan sammenligne en klasse som en plantegning for objekter.

Oppgave 1.3:

Hvis en klasse er plantegningen, er et objekt det ekte produktet, et objekt holder på de samme attributtene til klassen, men har sine egne verdier for de (som i de forrige eksemplene).

Oppgave 2: BlackJack

For å starte, begynte jeg med å tenke på hvordan jeg skulle håndtere kortene. Skulle det være en liste med objekter inni? Skulle det være randint? Blant mange andre løsninger. Til slutt bestemte jeg meg for å bruke en list, som inneholder alle kortene som objekter, med verdi og type kort som verdier i objektet. Da bare gjensto det å lage denne listen. Først prøvde jeg å gjøre det sånn her, der "rank" står for verdi, og "suit" står for type:

```
class Card:
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

card0 = Card("Ace", "Spades")
card1 = Card("Ace", "Clubs")
card2 = Card("Ace", "Diamonds")
card3 = Card("Ace", "Hearts")
card4 = Card("Two", "Spades")
card5 = Card("Two", "Clubs")
```

Jeg bestemte meg fort for at dette var et ork, så jeg bestemte meg for å gjøre det på en annen måte. Jeg bestemte meg for å bruke en for-løkke til å lage alle kortene mine i lista, jeg bestemte meg også for å bruke en klasse for å lage kortstokken, fordi jeg trengte å bli vant til klasser. Etter jeg hadde skrevet alt, ble det slik:

```
class Card:
    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit

class Deck:
    def __init__(self):
        self.cards = []
        suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
        ranks = ['Ace', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'Jack', 'Queen', 'King']
        for suit in suits:
            for rank in ranks:
                self.cards.append(Card(rank, suit))
```

Jeg tenkte at dette var en ganske god løsning, fordi da kan jeg lett lage flere bunker hvis jeg trenger det, og jeg kan lett blande kortene ved å gjøre:

```
deck = Deck()
random.shuffle(deck.cards)
```

Det neste jeg gjorde var å definere en funksjon for å trekke kort, siden det er noe man gjør mye av i Blackjack. Så jeg skrev denne funksjonen:

```
def draw_card(x):
    if x:
        return x.pop()
    else:
        return None
```

Denne funksjonen vil da returnere det siste kortet i listen, og fjerne det fra listen.

Etter dette, måtte jeg sjekke om alt dette funket. Så jeg satte opp en kjapp test, som skulle trekke et kort, så printe det:

```
card = draw_card(deck.cards)
print(f'Drawn card: {card}')
```

Og resultatet ble:

```
Drawn card: <__main__.Card object at 0x000001A49BB32090>
```

Dette er lite forståelig, så jeg ville forandre på det. Etter å ha sett rundt etter en løsning, fant jeg en. Jeg forandret på __init__-koden til Card-klassen slik:

```
class Card:
    card_count = 0

    def __init__(self, rank, suit):
        self.rank = rank
        self.suit = suit
        self.name = f'Card{Card.card_count}'
        Card.card_count += 1

    def __str__(self):
        return self.name
```

Kort forklart, det fikset det, sammen med:

```
print(f"Drawn card: {card}: {card.rank} of {card.suit}")
```

Ble resultatene nå til:

```
Drawn card: Card5: 6 of Hearts
```

```
Drawn card: Card45: 7 of Spades
```

```
Drawn card: Card2: 3 of Hearts
```

Kortene får verdi fra en funksjon og et dict, her er dictet, funksjonen vises senere.

```
values = {
    'Ace': 1,  # Ess håndteres inni funksjonen
    'Two': 2,
    'Three': 3,
    'Four': 4,
    'Five': 5,
    'Six': 6,
    'Seven': 7,
    'Eight': 8,
    'Nine': 9,
    'Ten': 10,
    'Jack': 10,
    'Queen': 10,
    'King': 10
}
```

Så må jeg lage resten av strukturen, det skal være en hand for spilleren, og en hand for dealeren:

```
class Hand:
    def __init__(self, x):
        self.name = x
        self.hands = []
        self.value = int(0)
        self.has_ace = False
```

```
player = Hand("player")
dealer = Hand("dealer")
```

Det gjorde jeg ved å definere en klasse “Hand”, så lagde jeg to “Hand”-objekter, en for spilleren, og en for dealeren. Så viste det seg at jeg ikke skrev like fort på denne siden som jeg gjorde på koden min, så jeg må kjapt gå gjennom det jeg gjorde på resten av koden.

Hand fikk fort en til attribut:

```
class Hand:
    def __init__(self, x):
        self.name = x
        self.hands = []
        self.value = int(0)
        self.result = ""  # må lagre verdier,
        self.has_ace = False
```

Hand.result brukes for å lagre resultatet av spillet, noe som er nødvendig for å håndtere utfallet

De faste variablene defineres på starten av koden, og ser sånn her ut:

```
chips = 10  # disse kunne ha blitt flyttet
wager = 0
gameOver = False
player_played = False
idealdeck = Deck()
playdeck = list(idealdeck.cards)
outcome = ""  # lagringggggg
player = Hand("player")
dealer = Hand("dealer")
```

Selveste spillet kjører inni en while loop, som ser sånn her ut:

```
while not gameOver:  # gameloop
    gamesetup()
    player.result = player_turn()
    dealer.result = dealer_turn()
    outcome = game_outcome()
    update_chips()
    results(chips)
    check_if_game_over(chips)
    choice = play_again()
    if choice == "playagain":
        continue
    elif choice == "notagain":
        break
    else:
        Exception(
            "something has gone REALLY wrong here, please restart the program")
```

Alle funksjonene som jeg har skrevet, blir direkte eller indirekte called inni her, og her er de viktigste funksjonene, i ingen bestemt rekkefølge:

Check_if_game_over():

```
def check_if_game_over(x):
    global gameOver
    if x <= 0:
        print("You are destitute! You lose........")
        gameOver = True
```

Lett, sjekker hvis du har 0 eller mindre (for sikkerhets skyld) chips, hvis du har det, taper du alt, og får ikke spille igjen

Play_again():

```
def play_again():
    while not gameOver:
        # GPT ga meg ideen om å legge til lower for å gjøre alt smått
        choice = input(
            "do you want to play again? 'y' for yes, 'n' for no.").lower()
        if choice == "y":
            print(
                "You have chosen to play again. the code will now reset your hands and
            return "playagain"
        elif choice == "n":
            print("You have chosen to not play again. Au revoir....")
            return "notagain"
        else:
            print("Invalid input. Please enter 'y' for yes or 'n' for no.")
```

Hvis gameOver er ikke sann, vil den gi spilleren valget å spille igjen.

Gamesetup():

```
def gamesetup():
    global playdeck
    global player_played
    player.hands.clear()
    dealer.hands.clear()
    player.has_ace = False
    dealer.has_ace = False
    player_played = False
    playdeck = list(idealdeck.cards)
    random.shuffle(playdeck)
    draw_to_hand(draw_card(playdeck), player)
    draw_to_hand(draw_card(playdeck), player)
    draw_to_hand(draw_card(playdeck), dealer)
    draw_to_hand(draw_card(playdeck), dealer)
    chipswager(chips)
    showhand(player)
    showhand_dealer(dealer)
```

Denne funksjonen er den som setter opp starten av spillet, så den gjør alt klar for at spilleren skal spille. Her ser man også at decket som faktisk blir brukt er ikke orginal-decket, men en kopi av det som stokkes umiddelbart etterpå. Dette gjør det slik at kortstokken vil oppføre seg som en ekte stokk, dvs at når et kort, si 3 kløver, blir trukket, kan man ikke trekke det igjen. Denne funksjonen inneholder også den som lar spilleren vedde.

Chipswager():

```
def chipswager(x):
    global wager
    while True:
        wager = int(
            input(f"You have {x} chip(s) remaining, how many do you want to bet?"))
        if wager > x:
            print("You can't bet more than you have!")
        elif wager < x:
            print(f"You have decided to bet {wager} chip(s)")
            break
        elif wager == x:
            print("All in? Gutsy.")
            break
```

Her får spilleren velge hvor mye de vil vedde. Simpelt som det, selv om jeg gjorde det så fancy som jeg kunne.

Showhand & showhand_dealer():

```
def showhand(x):#printer hånden
    print(f"The {x.name}'s hand consists of the following cards: ")
    for card in x.hands:
        print(f"{card.rank} of {card.suit}")
    print(f"For a combined value of: {sum_hand(x)}")


def showhand_dealer(x):#for å vise dealeren sitt synlige kort
    print(f"The {x.name}'s visible card is: ")
    d_card = x.hands[0]# chat hjalp meg her
    if d_card:
        print(f"{d_card.rank} of {d_card.suit}")
    print(f"For a value of: {sum_hand_dealer(x)}")
```

Simpelt som det, showhand printer hånda, og teller verdien, mens showhand_dealer gjør det bare på det første kortet, som simulerer at dealeren bare har ett synlig kort.

Sum_hand & sum_hand_dealer():

```python
def sum_hand(x):  # chatgpt hjalp meg her
    x.value = 0
    x.has_ace = False
    for card in x.hands:
        x.value += rank_value(card.rank)
        if card.rank == "Ace":
            x.has_ace = True
    if x.has_ace and x.value <= 11:  # sjekker om ess må være 11 eller 1
        x.value += 10
    return x.value


def sum_hand_dealer(x):  # chatgpt hjalp meg her
    x.value = 0
    x.has_ace = False
    d_card = x.hands[0]
    if d_card:
        x.value += rank_value(d_card.rank)
        if d_card.rank == "Ace":
            x.has_ace = True
    if x.has_ace and x.value <= 11:  # sjekker om ess må være 11 eller 1
        x.value += 10
    return x.value
```

Her beregnes verdien av kortstokken, ess behandles her også denne funksjonen forandrer hand.value verdien til hand-klassen, verdien nullstilles på starten av funksjonen fordi hvis den ikke hadde gjort det, ville den lagt til verdien av “nå-hånden” med “da-hånden”

Sum_hand_dealer gjør samme greia, men bare med det første kortet i hånda, slik at den viser bare verdien til det synlige kortet.

Draw_to_hand():

```python
def draw_to_hand(x, y):
    if x:
        y.hands.append(x)
```

Enkelt som det. Tar return-verdien fra draw_card, og appender en av hendene med det.

Player_turn():

```
def player_turn():
    global gameOver
    global player_played
    global dealer
    if len(player.hands) == 2 and sum_hand(player) == 21:
        print("Player has a Blackjack! Now to see what the dealer has.....")
        player_played = True
        return "blackjack"
    while player_played == False:
        decision = input(
            "Do you want to hit or stand? Enter 'h' for hit or 's' for stand: ").lower()

        if decision == "h":
            # får kort
            draw_to_hand(draw_card(playdeck), player)
            showhand(player)
            showhand_dealer(dealer)

            # ambasing
            if sum_hand(player) > 21:
                print("Player busts! You lose.")
                player_played = True
                return "bust"
        if len(player.hands) > 2 and sum_hand(player) == 21:
            # lar deg ikke throwe
            print("Player has 21!")
            player_played = True
            return ("stand")
        elif decision != "h" and decision != "s":
            print("Invalid input. Please enter 'h' for hit or 's' for stand.")
        elif decision == "s":
            player_played = True
            return ("stand")
```

Player_turn er kanskje den mest tungvinte funksjonen, så jeg skal forklare den i dybde:

Først, denne funksjonen er ment til å returne en string, som lagres i player.result. Player_played er en bool som sjekker om spilleren har gjort trekket sitt. Jeg fant ut at denne var viktig, fordi ellers vil den konstant spørre spilleren om flere trekk, selv om spilleren allerede har gjort trekket sitt

Første if-statement sjekker om spilleren har blackjack, og hvis de har det, returner funksjonen “blackjack” til player.result, og lar ikke spilleren gjøre noe.

While løkka er der spilleren får gjøre trekkene sine. Den kjører bare hvis player_played er false, så den vil stoppe når spilleren er ferdig med trekket sitt.

Så tar koden spilleren sitt valg med decision-inputtet, og vil da enten kjøre hit eller stand biten av kode. Hvis spilleren hitter, og ikke buster, får de lov å velge på nytt, helt til de enten buster, eller velger å stande.

Inni hit vil den først trekke et kort til spilleren sin hånd, så vise hånda til spilleren, så det synlige kortet til dealeren. Etter det vil den sjekke om spilleren har busta, hvis de har gjort det, vil den returne “bust” til player.result, og ikke la deg spille.

Etter det vil den sjekke om spilleren har nådd 21 uten å ha fått blackjack(for eksempel hvis de har 6,7,8 i hånda), hvis de har gjort det, vil den ikke la deg trekke noe mer, og vil stande for deg.

Så er det en if statement som skjekker om spilleren har skrevet noe feil inn, hvis det er tilfellet, vil den be spilleren om å skrive på nytt, helt til de får det riktig.

Helt til slutt vil den sjekke om spilleren valgte å stande, da vil den stande, og stoppe turen sin.

Dealer_turn():

```
def dealer_turn():
    if len(dealer.hands) == 2 and sum_hand(dealer) == 21:
        print("Dealer has a Blackjack!")
        return "blackjack"
    while sum_hand(dealer) < 17:
        # dealer spiller
        draw_to_hand(draw_card(playdeck), dealer)
    showhand(dealer)
    if sum_hand(dealer) > 21:
        print("Dealer busts!")
        return "bust"
```

Heldigvis er dealer_turn mye lettere. Denne funksjoner funker på samme måte som player_turn, dvs at den returner en string til .result, men denne gangen er det dealer.result. Her sjekker den først om dealer har blackjack, og hvis den har det, vil den returne blackjack. Her er det ikke nødvendig med en sjekk om dealeren allerede har spilt, fordi while-løkken stopper automatisk når dealerens hånd-verdi er over 17. Til slutt sjekker den om dealeren har busta, og håndterer det.

Game_outcome():

```
def game_outcome():  # om du vinner eller taper
    if player.result == "blackjack" and dealer.result == "blackjack":
        return "pushbyblackjack"
    elif player.result == "blackjack":
        return "blackjack"
    elif dealer.result == "blackjack":
        return "lossbyblackjack"
    elif player.result == "bust":
        return "lossbybust"
    elif dealer.result == "bust":
        return "win"
    elif sum_hand(player) == sum_hand(dealer):
        return "push"
    elif sum_hand(player) > sum_hand(dealer):
        return "win"
    else:
        return "loss"
```

Her sjekker den mellom player.result og dealer.result, og returner en string til outcome basert på det. Jeg strukturerte det slik at den sjekker for det sjeldneste resultatet (dobbel blackjack) først, så går det ned til nest sjeldneste osv. Til slutt sammenligner den verdiene til hendene på spiller og dealer, og sjekker hvem som vinner, hvis ingen andre spesielle tilfeller skjer først.

Update_chips():

```
def update_chips():
    global outcome
    global chips
    global wager  # mr worldwide
    if outcome == "win":
        chips += wager
    elif outcome == "loss":
        chips -= wager
    elif outcome == "lossbybust":
        chips -= wager
    elif outcome == "lossbyblackjack":
        chips -= wager
    elif outcome == "push":
        chips = chips
    elif outcome == "pushbyblackjack":
        chips = chips
    elif outcome == "blackjack":
        chips += (wager*2)
```

Her tar den outcome-et som ble bestemt i forrige funksjon, og utfører forandringen på chips-bestanden basert på det. Det er viktig å bemerke, at chips forandres ikke før denne funksjonen, så det er derfor matten ser sånn her ut. Hvis spilleren vinner, får de beholde hele chips-bestanden, og får innsatsen i tillegg. Hvis de taper er det chips – innsats, og hvis det er push, vil chips ikke forandre på seg, og hvis spilleren får blackjack, vil det bli chips + innsats*2.

Results():

```
def results(x):  # printer fancy ting i terminalen
    global outcome
    if outcome == "win":
        print("You win! Very nice!")
    elif outcome == "loss":
        print("You lose! Better luck next time....")
    elif outcome == "lossbyblackjack":
        print("You lose to blackjack! Unlucky! Better luck next time....")
    elif outcome == "lossbybust":
        print("You lose by busting! Unlucky! Better luck next time....")
    elif outcome == "push":
        print("Push! Nobody wins....")
    elif outcome == "push":
        print("Push by blackjack! Unlucky, or lucky, you decide....")
    elif outcome == "blackjack":
        print("You win by blackjack! Very nice!")
    print(f"As a result of the game, you now have {x} chip(s) remaining.")
```

Her vil den ta outcome-et og skrive noe tekst i terminalen om hvordan spillet endte opp. Her er det mange forskjellige utfall, fordi jeg ville legge til flavor i spillet.

Sånn. Det er alt det viktige i koden. Hvis jeg skulle ha gjort noe annerledes, ville jeg ha startet med å putte chips og wager inni hand-klassen, slik at man kunne lett ha gjort rede for flerspiller. Andre

forandringer på koden hadde vært å bytte om på game_outcome slik at “stand” resultatet faktisk gjør noe, og ikke bare er det for debug-grunner (jeg brukte det til å teste funksjonen ved å få den til å printe ut resultatet.).