Self-Review Questions

Self-review 6.1 Explain the roles of \( r \), \( \text{random} \), and \( () \) in:
\[
r.\text{random } ()
\]
\( r \) is an object, \( \text{random} \) is a method which is a member of the object \( r \) and \( () \) is the (empty) list of arguments passed to the method \( \text{random} \).

Self-review 6.2 What is an object?

Self-review 6.3 Why are classes important?

Self-review 6.4 What is a method?
A method is a piece of executable code, a bit like a function. Unlike a function, methods are members of objects and their first argument is always the object to which the method belongs (this argument is usually called \( \text{self} \)).

Self-review 6.5 Why is a method different than a function?

Self-review 6.6 Why is the method name \( \text{__init__} \) special, and for what reason(s)?

Self-review 6.7 What is an abstract data type?
An abstract data type is a type whose internal data structures are hidden behind a set of access functions or methods. Instances of the type may only be created and inspected by calls to methods. This allows the implementation of the type to be changed without changing any code which instantiates that type. Queues and stacks are examples of abstract data types.

Self-review 6.8 Why is there an apparent difference between the number of parameters in the definition of a method and the number of arguments passed to it when called?

Self-review 6.9 What does \( \text{self} \) mean? Is it a Python keyword?

Self-review 6.10 Is the method \( \text{bar} \) in the following code legal in Python?
class Foo:
    def __init__(self):
        self.foo = 1
    def bar(flibble):
        print(flibble.foo)

Self-review 6.11 What is operator overloading?

Operator overloading means creating a new definition of an operator (such as + or -) for a new datatype. For example the following class overloads the + and * operators, and the built-in function which tells Python how to print out a Vector object:

class Vector:
    def __init__(self, l):
        self.values = l
    def __add__(self, other):
        if len(self.values) != len(other.values):
            raise AssertionError, 'Vectors have different sizes.'
        v = []
        for i in range(len(self.values)):
            v.append(self.values[i] + other.values[i])
        return Vector(v)
    def __mul__(self, other):
        if len(self.values) != len(other.values):
            raise AssertionError, 'Vectors have different sizes.'
        v = []
        for i in range(len(self.values)):
            v.append(self.values[i] * other.values[i])
        return Vector(v)
    def __str__(self):
        return 'Vector ' + self.values.__repr__()

Self-review 6.12 Which operators do the following methods overload?

1. __add__
2. __eq__
3. __lt__
4. __or__
5. __ne__
6. __div__
7. __ge__

Self-review 6.13 What is the difference between a queue and a stack?

Self-review 6.14 Why is it usually thought to be a bad thing to use global statements?

Self-review 6.15 What does the __ mean in the name __foobar in the following class?
class Foo:
    def __init__(self):
        self.__foobar()
    def __foobar(self):
        print('foobar')

The double underscore means that the method __foobar can only be called by code within the class Foo. For example:

```python
>>> class Foo:
...     def __foobar(self):
...         print('foobar')
...     def foobar(self):
...         self.__foobar()
...
>>> f = Foo()
>>> f.foobar()
foobar
>>> f.__foobar()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: Foo instance has no attribute '__foobar'
>>> 
```

Self-review 6.16 What new facilities for encapsulation do classes offer?

## Programming Exercises

### Exercise 6.1
Create a class to represent a single die that can have any positive integer number of sides. This kind of die might be used when playing role-playing games (RPGs).

```python
class AnyDie:
    def __init__(self, sides):
        assert type(sides) == type(0)
        assert sides > 0
        self.sides = sides
        self.generator = random.Random()
        self.current = 1
        self.representations = map(lambda x: '*' * x, range(self.sides + 1))
    def __str__(self):
        return self.representations[self.current]
    def roll(self):
        self.current = self.generator.randint(1, self.sides)
        return self.current
```

### Exercise 6.2
Write a class to represent an RPG character’s money pouch. Money should be stored as an integer, with methods for adding money and removing money. The removing method should take a value as parameter. If there is enough money, the value is removed from the money in the pouch and True is returned. If there is not enough money, False is returned.
Exercise 6.3 Write a simple RPG character class. The character should have a name, a money pouch and an inventory. The name should be stored as a string, the money pouch should be an instance of the pouch from the previous exercise and the inventory should be a dictionary in which keys are item names and values are the number of items held by the player.

Ensure that there are methods for adding items to, and removing items from, the inventory.

There should be a __str__ method that returns something that can be printed. For example:

```
print playerA
```

Might display:

```
-------------------------
Duncan Disorderly
-------------------------
Money: 235 gold pieces
-------------------------
Knapsack contains:
Arrow: 12
Rubber Sword: 1
Felt tipped pen: 2
Imitation fur coat: 23
-------------------------
```

class Pouch :
def __init__ ( self , money ) :
    self.money = money
def add ( self , money ) :
    self.money += money
def remove ( self , money ) :
    if not ( self.money > money ) :
        return False
    self.money -= money
    return True
def __str__ ( self ) :
    return 'Money: ' + str ( self.money ) + ' gold pieces.'

class Character :
def __init__ ( self , name ) :
    self.name = name
    self.pouch = Pouch ( 0 )
    self.inventory = {}
def pickupItem ( self , item ) :
    if self.inventory.has_key ( item ) :
        self.inventory[item] += 1
    else:
        self.inventory[item] = 1
def dropItem ( self , item ) :
    if not self.inventory.has_key ( item ) : pass
    elif self.inventory[item] < 1: pass
    else : self.inventory[item] -= 1
def pickupMoney ( self , money ) :
self.pouch.add(money)
def dropMoney(self, money):
    self.pouch.remove(money)
def __str__(self):
    sep = '----------
    s = sep + self.name + \n' + sep + str(self.pouch) + \n' + sep + 'Knapsack contains:\n
    for key in self.inventory.keys():
        if self.inventory[key] > 0:
            s += ('\t' + key + ':' + str(self.inventory[key]) + \n')
    s += sep
    return s

Exercise 6.4 Implement the multiplication operation for the Matrix class.
The trick to implementing multiplication is to realize you need three, nested loops:

if len(self.data[0]) != len(m.data):
    raise ValueError, 'Matrices not of the suitable shapes for multiplication.'
    n = zeros(len(self.data), len(m.data[0]))
    for row in range(len(n.data)):
        for column in range(len(n.data[0])):
            for i in range(len(self.data[0])):
                n.data[row][column] += self.data[row][i] * m.data[i][column]
    return n

Exercise 6.5 Implement __getitem__ for the Matrix class so that we can rewrite the
drawTriangle function to work with a triplet of 2x1 matrices:

    def drawTriangle(coordinateMatrix):
        turtle.up()
        turtle.goto(coordinateMatrix[0][0][0], coordinateMatrix[0][0][1])
        turtle.down()
        turtle.goto(coordinateMatrix[1][0][0], coordinateMatrix[1][1][0])
        turtle.goto(coordinateMatrix[2][0][0], coordinateMatrix[2][1][0])
        turtle.goto(coordinateMatrix[0][0][0], coordinateMatrix[0][1][0])

We can use __getitem__ in exactly the same way that we would use any list subscript. For example:

    >>> m = Matrix(2,2)
    >>> m[(0,0)]
    0.0
    >>>

Exercise 6.6 Write a class Account that stores the current balance, interest rate and
account number of a bank account. Your class should provide methods to withdraw, deposit and add interest to the account. The user
should only be allowed to withdraw money up to some overdraft limit. If an account goes overdrawn, there is fee charged.

Exercise 6.7 Write a small class to represent the light switch state machine from
Section ???. Provide a single method to change the state of the switch
and method called isOn which returns True if the switch is on and
False if it is off. Make sure you override the __str__ method so that
light switches can be printed to the console.
class Account:
    interest_rate = 0.05
    overdraft_limit = 500
    overdraft_charge = 50
    def __init__(self, number):
        self.number = number
        self.balance = 0
    def deposit(self, amount):
        if amount < 0:
            raise ValueError, "Can't deposit a negative amount of money."
        self.balance += amount
    def withdraw(self, amount):
        if amount < 0:
            raise ValueError, "Can't withdraw a negative amount of money."
        if amount > (self.balance + Account.overdraft_limit):
            raise ValueError, 'Out of credit.'
        if amount > self.balance:
            self.balance -= amount
            self.balance -= Account.overdraft_charge
        else:
            self.balance -= amount
    def __str__(self):
        sep = '------------------'
        s = '
Account
' + sep + '
' + 'Account number: ' + str(self.number) + '
' + sep + '
' + 'Overdraft limit: ' + str(Account.overdraft_limit) + '
' + sep + '
' + 'Overdraft charge: ' + str(Account.overdraft_charge) + '
' + sep + '
' + 'Account balance: ' + str(self.balance) + '
' + sep + '
' + return s

Exercise 6.8 Write a program which uses the Stack class. Your program should begin by printing a menu to the user:

1. Add new data to stack
2. Print stack
3. Remove datum from stack
4. Exit

You should allow the user to enter 1, 2, 3 or 4 to select their desired action and you should write code to implement the four possible options.

Exercise 6.9 Amend your program from Exercise 6.8 to use a Queue as the data structure used to store data in.

```python
menu = """1. Add new data to the queue.
2. Print queue.
3. Remove datum from the queue.
4. Exit.
"""

if __name__ == '__main__':
    q = Queue()
    while True:
        print menu
        m = 0
        while m < 1 or m > 4:
            m = input ('Enter: ')
if m == 1:
    datum = raw_input('Enter datum: ')
    q.add(datum)
elif m == 2:
    print 'Queue:', q, '
'elif m == 3:
    datum = q.remove()
    print 'Removed', datum, 'from the queue.
'elif m == 4:
    print 'Goodbye!' break

Exercise 6.10 A priority queue is an abstract data type similar to the queue introduced in Section ?? and Section ?? . A priority queue associates a priority with each stored item and always stores the items so that the elements with the highest priority are at the 'top' of the queue and are the first to be removed – i.e. the items in a priority queue are sorted by priority. Create a PriorityQueue class based on the Queue class:

1. The add method should take two parameters, an item to store and a priority, which should be an integer.
2. The add method should ensure that when new data is added to the priority queue, it is added as a tuple which contains both the data and its priority. Make sure that data is always stored in priority order.
3. The remove method should return queue items, not tuples – i.e. the priority associated with the returned item can be ignored.

Challenges

Challenge 6.1 Currently, the n-faced die (see Exercise 6.1) is unable to display its spots and must instead rely on displaying a number.

Your task is to write a function that returns a string representation of the die face for a given number. For example, this:

print makeFace (9)

might display:

***
***
***

The algorithm is up to you, but do remember about integer division and the remainder operator (%)

Now that you can make faces with an arbitrary number of spots, add this functionality to your n-sided die class.

Challenge 6.2 Extend your RPG character class to hold values for health points (HP), attack points (AP) and defence points (DP).
Add an attack method that takes a character instance as a parameter. This is your opponent.

If the character’s AP is greater than the opponent’s DP, the difference is subtracted from the opponent’s HP. So, if I attack with a power of 7 and my opponent has a defence power of 6, they lose 1 health point. If they have a defence power of 9, they sustain no damage.

Write a program that demonstrates your character class by creating two characters that take it in turns to bop each other until one of them runs out of HP.

```python
from pouch import Pouch
import random

class Character:
    def __init__(self, name):
        self.name = name
        self.pouch = Pouch(0)
        self.inventory = {}
        self.hp = 100
        self.ap = random.randint(1, 100)
        self.dp = random.randint(1, 100)
    def attack(self, enemy):
        if self.ap > enemy.dp:
            enemy.hp -= (self.ap - enemy.dp)
    def pickupItem(self, item):
        if self.inventory.has_key(item):
            self.inventory[item] += 1
        else:
            self.inventory[item] = 1
    def dropItem(self, item):
        if not self.inventory.has_key(item):
            return
        elif self.inventory[item] < 1:
            return
        else:
            self.inventory[item] -= 1
    def pickupMoney(self, money):
        self.pouch.add(money)
    def dropMoney(self, money):
        self.pouch.remove(money)
    def __str__(self):
        sep = '----------------
        s = sep + self.name + '\n' + sep
        s += 'Health:' + str(self.hp) + '\n'
        s += 'Attack:' + str(self.ap) + '\n'
        s += 'Defence:' + str(self.dp) + '\n'
        s += sep + str(self.pouch) + '\n' + 'Knapsack contains:\n
        for key in self.inventory.keys():
            if self.inventory[key] > 0:
                s += ('\t' + key + ': ' + str(self.inventory[key]) + '\n')
        s += sep
        return s

if __name__ == '__main__':
    player1 = Character('Player 1')
```
player2 = Character ( 'Player 2' )
print '***** Testing... Creating character\n'
print player1
print player2
print '***** Testing... Mortal Kombat!\n'
while player1.hp > 0 and player2.hp > 0 :
    # Player 1 always hits first -- grossly unfair :-(
    player1.attack ( player2 )
    player2.attack ( player1 )
    print player1
    print player2