Quiz 3

A. About structures

Functions bundle up executable statements under one name
Modules bundle up functions, statements, classes in one file
Classes bundle up functions and variables in one object

How they differ:

<table>
<thead>
<tr>
<th>Modules</th>
<th>Classes (objects)</th>
</tr>
</thead>
<tbody>
<tr>
<td>only one of each</td>
<td>many copies of each class</td>
</tr>
<tr>
<td>cannot contain other modules</td>
<td>can contain other classes</td>
</tr>
<tr>
<td>import</td>
<td>make a new object</td>
</tr>
<tr>
<td>you can import all and do</td>
<td>you always need the DOT notation</td>
</tr>
<tr>
<td>away with module DOT</td>
<td></td>
</tr>
<tr>
<td>no self variable</td>
<td>self variable needed</td>
</tr>
</tbody>
</table>

A package is just a directory containing 1 or more modules.

But you can include __init__.py in the directory which will initialize the modules.

```
instance = object

import sys
import ParserHelper

ParserHelper.__file__

File path name — tree walk to a file or dir

C:\Users\meyer\CSCI12\LABS\Project2\file.py

full path partial path

c: escape /home/meyer/CSCI12/LAB/YProject2/file.py

char \n "t" \b 'g' \n
look & feel
```
B. Recursion

A function that calls itself, but cannot go on forever or it will be an infinite loop

Divide and conquer strategy (divide et impera)

Every recursive function can be implemented with a loop but oftentimes the recursive definition is so much cleaner. Will need a stack to implement.

Data structures are defined recursively:

A list is either
0. an empty list with no elements
1. a single element concatenated into a list

That underlined term is the key. That list may be either case 0 or 1.

List examples:

[] 'a' concatenated with []
'[a,b]' 'e' concatenated with [b] where [b] is b concatenated with []

etc.

A binary tree is either
0. a single node
1. a single node with two children: A left child which is a tree
   A right child which is a tree

Quite often a recursive data structure and a recursive algorithm go hand in hand.

Other recursion:

\[ n! = \begin{cases} 
 0 & \text{if } n = 0 \text{ or } 1, \text{ then } n! = 1 \\
 0 & \text{otherwise } n! = n \times (n-1)! 
\end{cases} \]

\[ \text{def } \text{fact}(n): \]
\[ \text{if } n < 2: \]
\[ \text{return } 1 \]
\[ \text{else:} \]
\[ \text{return } n \times \text{fact}(n-1) \]

len(somelist) =
- if somelist is [], then 0
- otherwise 1 + len(somelist with the first element removed)

An arithmetic expression is:
- a single number or a variable
- matching parentheses around an arithmetic expression
- two arithmetic expressions with an operator in between

\[ \text{fact}(5) = 5 \times 4 \times 3 \times 2 = 120 \]
\[ 5 + 2 \]

\[ 5 \times 4 \times 3 \times \text{fact}(2) \]
\[ 5 \times 4 \times 3 \times \text{fact}(3) \]
\[ 5 \times 4 \times 3 \times \text{fact}(4) \]
\[ 5 \times 4 \times 3 \times \text{fact}(2) \]
\[ 3 \times 2 \times 4 \times (6) \]
There's always
1. a base case: the simplest possible ground case, recursion ends here
2. a continuation case: where the recursion is invoked

Fibonacci numbers: See wikipedia article about their importance

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
</tr>
</tbody>
</table>

\[ F_1 = 1, F_2 = 1, F_n = F_{n-1} + F_{n-2} \]

def fib(n):
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```python
def empty(L): return True if L else False
def head(L): return first thing in list
def tail(L): return rest of list except head
def insert(x, L): new list of [x, in] L

def length(L):
    if empty(L):
        return 0
    else:
        return 1 + length(tail(L))
```

1. Recursive function calls itself
2. Has 2 part structure
3. It calls itself on a smaller instance of itself
Functional Programming
includes recursion
excludes explicit loops
"changing a var's value"

Pstruct: single statement (assign, print)
or
1) if cond:
Pstruct
else:
Pstruct

2) while cond:
Pstruct

3) try:
Pstruct
except:
Pstruct
Exercise

\[ x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot x^{n-1} & \text{else} \end{cases} \]

\[ x^5 = x \cdot x^4 \]

```python
def pow(x, n):
    if n == 0:
        return 1
    else:
        return x * pow(x, n-1)
```

\[ \text{exp}(n) = e^n \]
def sort(L):
    if length(L) < 2:
        return L
    else:
        # split L into L₁ and L₂
        L₁ = sort(L[:n])
        L₂ = sort(L[n:])
        return merge(L₁, L₂)

def merge(L₁, L₂):
    return L₁ + L₂

def length(L):
    return [ ... ]

def sort(L):
    # L₁ = L[0:n]
    # L₂ = L[n:]
    L₁, L₂ = split(L)
    return merge(sort(L₁), sort(L₂))

L = [8, 4, 10, 4, 8, 16]
L₁ = sort([7, 6, 9])
L₂ = sort([5, 4])
L = linear_merge(L₁, L₂)
L = [4, 5, 6, 8, 9, 10]
**Recursive bubble sort**

```python
def insert_sorted(x, L):
    if len(L) == 0:
        return [x]
    if x < L[0]:
        return [x] + L
    else:
        return [L[0]] + insert_sorted(x, L[1:])

ox = [2, 5, 7, 9, 15, 16, 17]
print(insert_sorted(20, x))

def sort(L):
    if len(L) < 2:
        return L
    return insert_sorted(L[0], sort(L[1:]))

print(sort([5, 2, 7, 4, 9, 10, 15, 8]))
```

**Many other kinds of sorts:**

Mergesort:
split the list in half
call mergesort on each half and save the new sorted sublists
merge the sorted sublists

**Game-playing**

Backtracking – all possible solutions