Review

- Exceptions Handling
- Exam 2 Topics
- String Manipulation Review
- Data Structures Review

Assignment 7 – Common data file problems

1. Disallowed characters in numeric strings
   - Allowable characters in strings convertible to a numbers include:

     `<digits> . + - E`
   - Disallowed characters include:

     `$ , % …`

   Delete all disallowed characters from numeric strings in your data file

Assignment 7 – Common data file problems

2. A “,” character in a CSV file string
   - Excel surrounds strings containing a comma with quotations. But, Python does not honor this notation.
     - Excel will export a cell containing the data:

       `Mars, now with organic compounds`

       as:

       `"Mars, now with organic compounds"`

   - Python will ignore the quotes and split on the embedded comma, causing problems

Replace or delete all commas contained within strings in a CSV file

Assignment 7 – Common data file problems

3. An initial row of column headers
   – Many data sets start with a row of strings that identify columns. If you read this row as data and attempt to convert a row header to a number … runtime error

Remove any header row from your data file, especially if you have numeric data

Exam 2 Topics

1. String Manipulation
   - len(), strip(), split(), join()
2. Data Structures
   - lists, dictionary, combinations
3. Transformations
   - pushMatrix(), popMatrix(), translate(), rotate(), scale()
4. Image Processing
   - loadPixels(), updatePixels(), getPixel(), setPixel(), image()
5. Functions
   - defining, calling
6. Debugging
7. Recursive Functions
   - Designing: conditional, base case, recursive function call
8. Inheritance
   - subclassing, methods and instance variable overriding

Recursive Functions

Must have…

1. A recursive call
   - A branch that (ultimately) leads to at least one self-call
2. Base case
   - A branch that stops recursion – no self-call
3. A conditional to tell the difference between 1 & 2

Often have…

- A parameter that changes, allowing the base case to be identified

$5! = 5 \times 4 \times 3 \times 2 \times 1$

$4! = 4 \times 3 \times 2 \times 1$

$5! = 5 \times 4!$

$N! = N \times (N-1)!$

Factorial can be defined in terms of itself

$4! = 4 \times 3 \times 2 \times 1 = 24$

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

$5! = 5 \times 4!$

$4! = 4 \times 3!$

$3! = 3 \times 2!$

$2! = 2 \times 1$

Factorial – Recursive Implementation

```
1.   def factorial( i ):

2.       if i == 0:
3.           return 1
4.       else:
5.           return i*factorial(i-1)

6.   f = factorial(10)
7.   print(f)
```

Trace it.

factorial.py

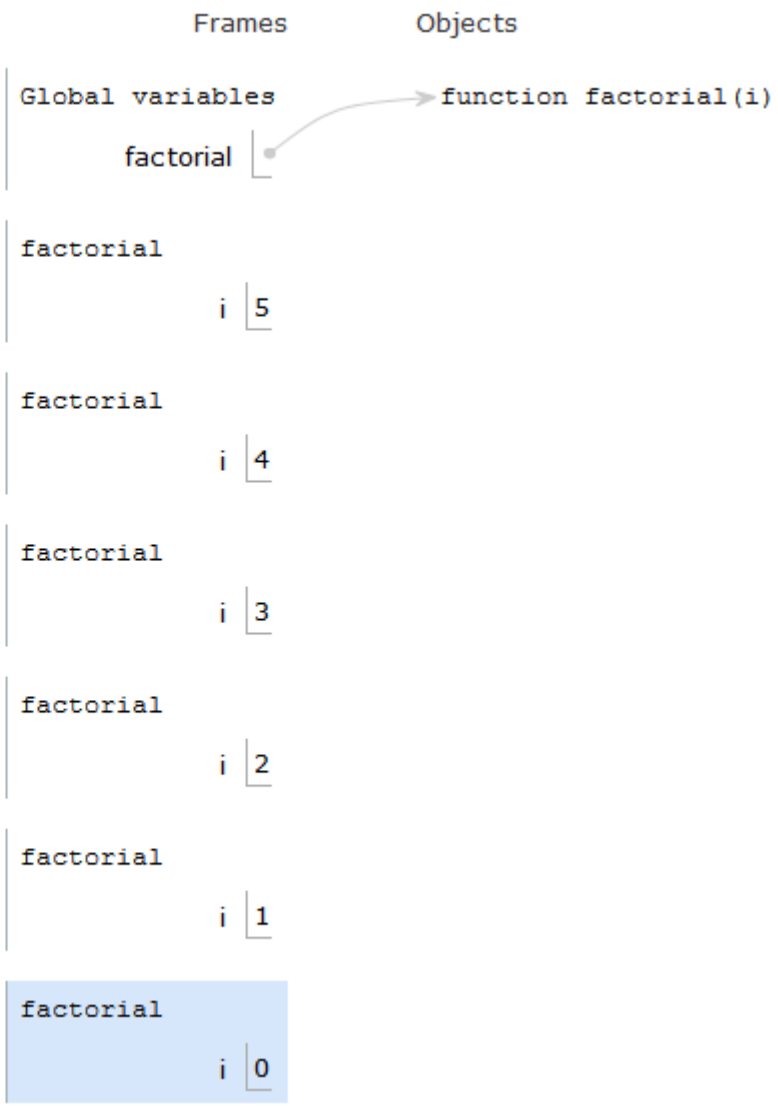
```
1  def factorial( i ):
2      if i == 0:
3          return 1
4      else:
5          return i*factorial(i-1)
6
7  f = factorial(5)
8  print(f)
```

Edit code

<< First | < Back | Step 14 of 21 | Forward > | Last >>

that has just executed
line to execute

output:



factorial.py

The Call Stack keeps track of …

1. all functions that are suspended, in the reverse order in which they were suspended (LIFO)
2. the point in the function where execution should resume after the invoked subordinate function returns
3. a snapshot of all variables and values within the scope of the suspended function so these can be restored upon continuing execution

```
# fibonacci.py
# Fibonacci sequence 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
# Compute Fibonocci sequence recursively

# Compute and return the nth Fibonacci number
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        f = fibonacci(n-1) + fibonacci(n-2)
        return f

f = fibonacci(12)
print(f)
```

fibonacci.py

Recursive Function Model

```
# Model of a recursive function

def recursiveFunction( args ):

    if base_case_condition :
        return base_case_value (if any)

    else:
        modified_args = (some modification of args)
        f = recursiveFunction( modified_args )
        return f

f = recursiveFunction( args )
print(f)
```

Recursive Object Construction

```
# fractalTree.py
from Processing import *
window(600, 600)
background(255)

class FractalTree:
    def __init__(self, length, depth):
        self.length = length
        self.left, self.right = None, None

        # Not leaf. Grow further.
        if depth > 1:
            depth -= 1
            self.left  = FractalTree(0.6*length, depth)
            self.right = FractalTree(0.6*length, depth)

…

f = FractalTree(-200, 10)

translate(300, 600)
f.draw(50)

print(f.countBranches())
```

fractalTree.py

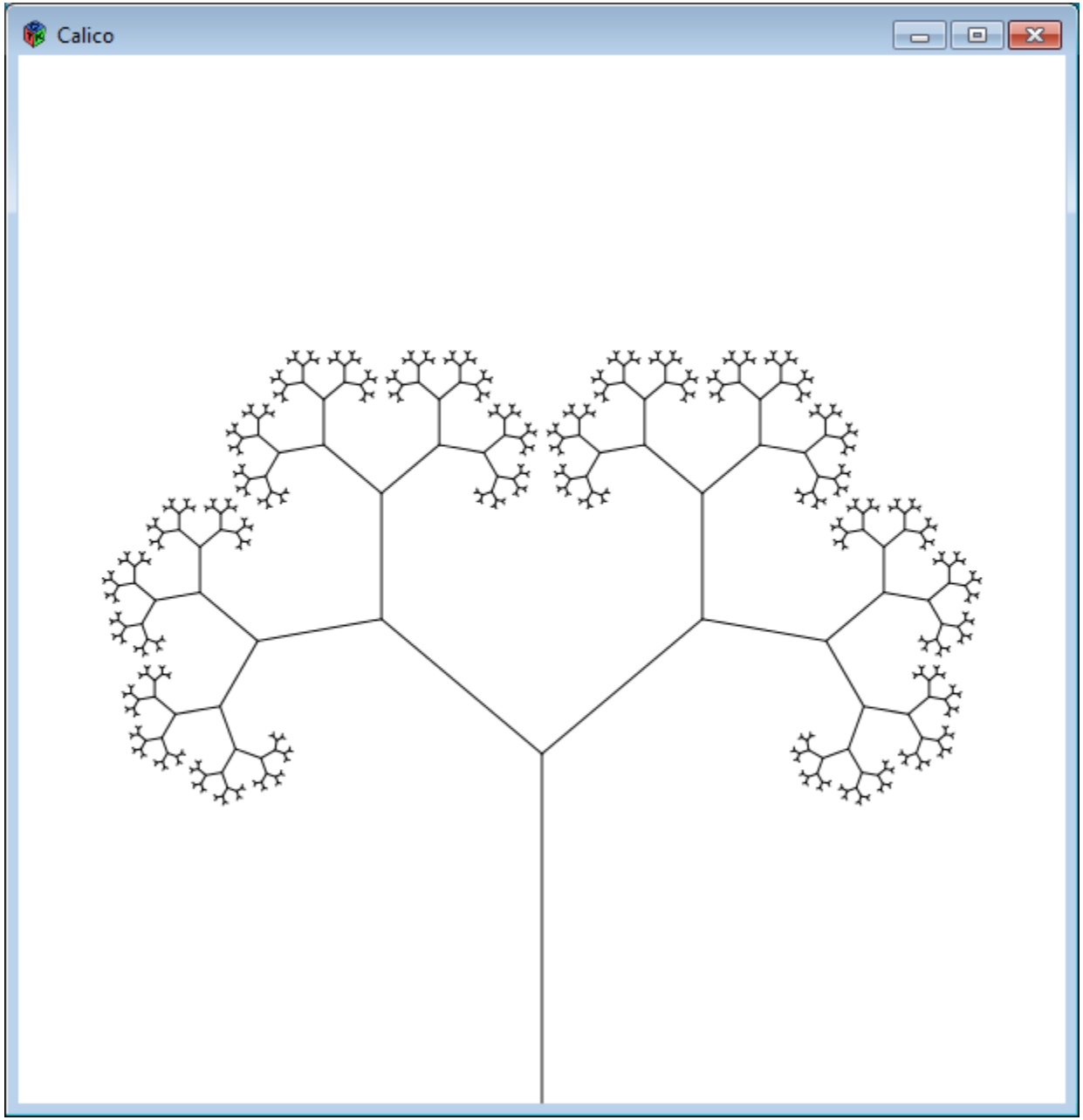
Recursive Object Rendering

```
# Recursively draw tree
def draw(self, angle):
    stroke(0)
    line(0, 0, 0, self.length)
    if self.left != None and self.right != None:
        translate(0, self.length)
        pushMatrix()
        rotate( radians(angle) )
        self.left.draw(angle)
        popMatrix()
        pushMatrix()
        rotate( radians(-angle) )
        self.right.draw(angle)
        popMatrix()

# Recursively count the depth of the tree
def countBranches(self):
    if self.left == None or self.right == None:
        return 1
    else:
        countLeft = self.left.countBranches()
        countRight = self.right.countBranches()
        return 1 + countLeft + countRight
```

fractalTree.py

Recursive Object Construction



Creating Recursive Functions

1. Start the function (`def` statement)
   - Include any necessary arguments
2. Set up a conditional to detect the base case
   - Usually by testing the argument value
3. Implement the base case
   - Can be as simple as a `return` statement
4. Implement the recursive case(s)
   - Usually involves modifying the arguments to approach the base case

Recursion – Practice

- Write a function that takes one integer argument and returns the sum of all integers down to 0

```
# recursion1.py
def recursiveSum( n ):
    if n <= 0:
        return 0
    else:
        tsum = n + recursiveSum( n - 1 )
        return tsum


print( recursiveSum( 10 ) )
```

Recursion – Practice

Euclid's Greatest Common Divisor algorithm (remainder version) says that:

`GCD( a, b ) is a if b == 0, and GCD( b, a % b) otherwise.`

Create a recursive implementation.

```
# recursion2.py
def GCD( a, b ):
    if b == 0:
        return a
    else:
        return GCD( b, a % b )

print( GCD( 75, 50 ) )
```

Recursion – Practice

Write a recursive function that recursively draws circles at the edges of a parent circle, half its diameter, all the same y-value. Stop at a radius of 1.

```
# recursion3.py
from Processing import *
window(600, 400)
ellipseMode(CENTER)
noFill()

def circles( x, rad ):
    if rad <= 1:
        return
    else:
        ellipse( x, 200, 2*rad, 2*rad )
        rad2 = 0.5*rad
        circles( x - rad, rad2 )
        circles( x + rad, rad2 )

circles( 300,  100 )
```

Recursion – Practice

Write a recursive function that recursively draws squares at the corners of a parent square, half its size. Stop at a size of 20.

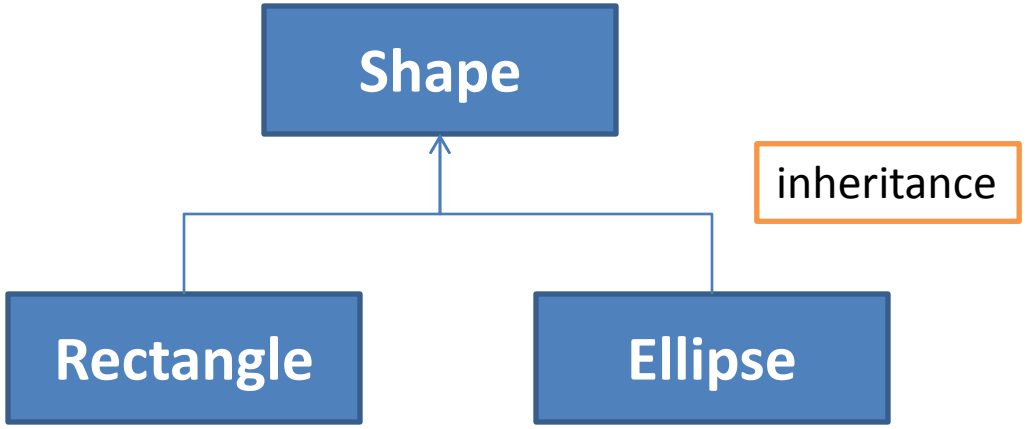
```
# recursion4.py
from Processing import *
window( 500, 500 )
rectMode(CENTER)

def squares( x, y, s ):
    if s <= 20:
        return
    else:
        rect(x, y, s, s)
        s2 = 0.5*s
        squares( x-s2, y-s2, s2)
        squares( x-s2, y+s2, s2)
        squares( x+s2, y-s2, s2)
        squares( x+s2, y+s2, s2)

squares( 250, 250, 200 )
```

Inheritance

We can set up an explicit relationship between Rectangle and a new class called Shape, and between Ellipse and Shape, called ***Inheritance.***



*This will automatically cause Shape **variables** and **methods** to be automatically accessible by Rectangle and Ellipse.*

Base Class : Shape
Child Class: Rectangle, Ellipse

Inheritance Relationship

- Set up between classes by adding the base class name in parentheses after child class name
- Optionally, invoke base class constructor with self parameter if child class

```
# Rectangle Class - After
class Rectangle(Shape):
    def __init__(self, pts):
        Shape.__init__(self, pts)
```

The Power of Inheritance

- A new behavior can be added easily to all child classes by adding once to a common base class
- A common behavior of all child classes can be modified easily by making changes to a base class
- Entirely new child classes can be created by declaring only how it differs wrt the base class

Inheritance - Practice

Create an inheritance hierarchy made up of three classes: Insect, Spider, and Butterfly.

- Most Insects have 6 legs, no wings, and crawl in response to a move() command.
- Spiders have 8 legs.
- Butterflies have wings, and flap in response to a move() command.

Inheritance – Base Class

```
# inheritance1.py

class Insect:
    def __init__(self):
        self.numLegs = 6
        self.hasWings = False

    def move(self):
        print('crawl crawl')
```

Inheritance – Spider Child Class

```
# inheritance1.py

class Insect:
    def __init__(self):
        self.numLegs = 6
        self.hasWings = False

    def move(self):
        print('crawl crawl')


class Spider(Insect):
    def __init__(self):
        Insect.__init__(self)
        self.numLegs = 8
```

Inheritance – Butterfly Child Class

```
# inheritance1.py

class Insect:
    def __init__(self):
        self.numLegs = 6
        self.hasWings = False

    def move(self):
        print('crawl crawl')


class Butterfly(Insect):
    def __init__(self):
        Insect.__init__(self)
        self.hasWings = True

    def move(self):
        print('flap flap')
```

Inheritance – Using Child Classes

```
# inheritance1.py
class Insect:
    def __init__(self):
        self.numLegs = 6
        self.hasWings = False

    def move(self):
        print('crawl crawl')

class Spider(Insect):
    def __init__(self):
        Insect.__init__(self)
        self.numLegs = 8

class Butterfly(Insect):
    def __init__(self):
        Insect.__init__(self)
        self.hasWings = True

    def move(self):
        print('flap flap')
```

```
# What is printed?
I = Insect()
S = Spider()
B = Butterfly()

print( "Insect:" )
print( "Has %s legs" % I.numLegs )
print( "Has wings: %s" % I.hasWings )
I.move()

print( "Spider:" )
print( "Has %s legs" % S.numLegs )
print( "Has wings: %s" % S.hasWings )
S.move()

print( "Butterfly:" )
print( "Has %s legs" % B.numLegs )
print( "Has wings: %s" % B.hasWings )
B.move()
```