CSSE 120 – Introduction to Software Development

Exam 1 – Format, Topics and Sample Problems

Fall term, 2010-2011

Format: The exam will have two sections:

- **Part 1: Paper-and-Pencil.**
    - **External resources allowed**: Your own textbook and a one-page (back and front) "cheat sheet" with whatever you want on it.
        - You should create your own cheat sheet (working with someone else is fine) as that will maximize both your learning and your score on the exam.
        - If you don't have a textbook or forget yours, you may briefly borrow the textbook that we bring to the exam (sharing it with anyone else who needs it).
    - **Points**: Approximately 50 of 100.
- **Part 2: On-the-computer.**
    - **External resources allowed**: Any printed or handwritten materials you choose to bring, your own computer, your own SVN repository, and any material directly reachable from the CSSE 120 Angel and web sites for this term.
        - For example, you may bring your own or other textbooks, notes, summary or study guides, and printouts from anywhere on the Internet.
        - During the exam, you may access anything on your computer.
        - During the exam, you may use the Internet ONLY to access your own SVN repository and anything directly reachable from this term's Angel course or from this term's web sites: www.rose-hulman.edu/class/csse/csse120/201110 and www.rose-hulman.edu/class/csse/csse120/201110robotics.
    - **Points**: Approximately 50 of 100.

During the exam, you may NOT communicate in any way with anyone other than your instructors or their assistants. Please turn off email, chat, etc. during the exam.

Topics:

- The ideas presented in **Sessions 1 through 10** (and *not* those introduced in Session 11).
- **Chapters 1 through 7** of your textbook.
    - Also, sections 8.1 and 8.4 may be helpful: the former reviews ***for*** loops and the latter discusses Boolean operators (***and***, ***or***, ***not***) and truth tables.

What you should be able to do: In the following, we have tried to list ***everything*** **that you might be expected to demonstrate on the exam**, plus ***sample problems*** for as much of that as practical. But please note:

- This is not a *contract*; it is only our *best-effort* to list everything you might be expected to demonstrate on this exam.
- The sample problems that will follow are just that – *samples*. **The problems on the exam may look both *similar to* and somewhat *different from* the samples.**

For the *Paper-and-Pencil* portion of Exam 1, students should be able to:

> ***Sample problems*** of each of these items appear ***later in this document***. If you don't understand what an item here is asking, see if the example problem clarifies matters for you.

1. **Read short snippets of code:**
   a. ***Trace short snippets of code*** (less than, say, 10 lines or so) and show ***what gets printed*** or the ***values of indicated expressions***.

   > **Expect *several questions* of this form.**

   b. ***Indicate errors*** in short snippets of code:
      - Syntax errors: something obviously wrong or missing
      - Semantic errors: does *such-and-such*, should do *so-and-so*

> The *next page* has ***a list of concepts*** that you might see in these snippets of code that you *read* and *write*.

2. **Write short snippets of code**, especially:
   a. ***range*** **expressions** for common ranges
   b. **loops** that generate/print simple sequences
   c. **function definitions**, including those that take parameters and/or return values
   d. **function calls,** including follow-up code that uses a returned value

3. **Explain important concepts of software development**, chosen from:
   a. The difference between a *specification* and an *implementation*, and what a *specification* of a function should include
   b. *Documentation*: how and why we put *internal comments* and *documentation strings* (*doc-comments*) in our programs
   c. *Software development tools*: what is provided by a typical, modern:
      - Integrated Development Environment (IDE)
      - Version control system
      - Debugger
   d. *Software development processes*:
      - What are some important *phases* of software development?
      - What is *procedural decomposition*, and why are functions useful?
      - What is a *compiler*? An *interpreter*?
      - What is an *algorithm*?

   > While you might see some problems of type #3, don't expect a lot of such questions and don't expect them to be deep.
   >
   > A simple understanding of these concepts is adequate.

   e. Key ideas of *object-oriented programming*, including:
      - What makes objects different from traditional data types, namely: ***objects know stuff*** (stored in *instance variables*) and ***can do stuff*** (via *methods*)
      - Why object-oriented programming is valuable
      - The difference between a *function* and a *method*, and the different notations for invoking them
      - The difference between an *object* and a *class* to which that objects belongs
      - The difference between *accessor* and *mutator* methods
   f. What is the difference between the **`int`** and **`float`** data types? What are the limitations of each? When should use one and when the other?
   g. The implications of the fact that variables in Python are *names* that point to *values* in memory, that is, variables are *references* to their values

Concepts that you might see on *code* that you *read* and *write* include:

Sample problems of each of these items appear ***later in this document***. If you don't understand what an item here is asking, see if the example problem clarifies matters for you.

1. **Variables and assignment,** including simultaneous assignment (`x, y = …, …`) and operator assignment (`sum += …`)
2. **Data types**: *int*, *float*, sequences (*lists*, *strings*, *tuples*, *range* expressions)
3. **Arithmetic and character expressions**, including those involving:
   - Operators: `+ - * / // % **`
   - Math functions: `abs cos sin pi sqrt`
   - Character functions: `ord chr`
4. The ***input*** function, including:
   - Providing a prompt
   - Converting an input string into a number (integer or floating-point) using ***int*** and ***float***
   - Stripping whitespace from the beginning and end of an input string (using ***strip***)
   - Splitting an input string into a list of strings (using ***split***) and then converting the strings in the list into appropriate types
5. The ***print*** function, including:
   - Printing on multiple lines or on the same line
   - Using a string's ***format*** method and associated format specifiers to do formatted output, especially: columns lined up on decimal points, centering
6. **Sequences**: *Lists*, *strings*, *tuples* and *range* expressions. Including:
   - *Indexing* and *slicing*, including negative indices. Accessing characters inside strings inside lists, etc.
   - The *len* function
   - Concatenation (`s1 + s2`) and duplication (`s * n`)
   - String methods like: ***capitalize count find format index join lower replace strip split title upper***
   - List methods like: ***append count index insert remove reverse sort***
7. **Definite loops**, including:
   - *Counted* loops through a ***range*** expression
   - Looping directly through a list or string
   - Looping through a list or string using its indices as generated by a ***range*** expression
8. **Functions and methods**, including:
   - Function *definitions*, including parameters
   - Function and method *calls*, including those with actual arguments
   - *Returning* a value from a function and capturing/using returned values
   - *Mutators* and mutable parameters
   - *Optional parameters* – defining, using
   - Functions that call functions
9. **Objects**, including statements that:
   - *Construct* an object
   - Apply a *method* to an object
   - Reference an *instance variable* of an object

   Zellegraphics as an example of *classes*, *constructors*, *methods* and *objects*
10. **Conditionals**, including:
    - The three forms:
      ```
      if
      if-else
      if-elif-elif...-else
      ```
    - *Relational operators* on numbers/strings: `<  >  <=  >=  ==  !=`
    - *Boolean operators*: `and  or  not`
11. ***import* statements**, in their three forms:
    ```
    import blah
    from blah import x, y, z
    from blah import *
    ```
12. **Reading and writing to files**:
    - *Opening* a text file for reading or writing. *Closing* a file.
    - Writing to a text file using *write*
    - Reading from a text file using the form:
      ```
      for line in file:
          ...
      ```

For the *On-the-Computer* portion of Exam 1, students should be able to:

Class A items: ***Study these items first*** and expect much of the test to focus upon them.

- Class B items come later in this list.

The ***List of Concepts*** from the *Paper-and-Pencil description* above applies to this portion of the exam as well.

> ***Sample problems*** of each of these items appear ***later in this document***. If you don't understand what an item here is asking, see if the example problem clarifies matters for you.

1. ***Write short programs and/or functions*** that are examples of the ***input/compute/output pattern***. Be able to:
   a. Use the `input` function to get input from the console, including:
      - Provide a prompt
      - Convert an input string into a number (integer or floating-point) using the `int` and `float` functions
   b. Use *variables* to store the input and perform numeric computations using:
      - Operators: `+` `-` `*` `/` `//` `%` `**`
      - Functions: `abs` `cos` `sin` `pi` `sqrt` `round`
   c. Use *print* to display results on the console

2. ***Define functions*** that have ***parameters*** and (possibly) ***return values***. Be able to:
   a. Write the `def` portion of a function definition, given (in ordinary English) the name of the function and a description of its parameters.
   b. Write the *function body*, using the *parameters* and other *local variables* as needed. Display an understanding of:
      - The fact that a parameter is a name for a value that comes into the function
      - The relationship of parameters and other local variables to variables with the same name outside the function
      - When and why to introduce local variables
   c. *Return* a value if called for by the problem

3. ***Call (invoke) functions***, both ordinary functions and *methods*, and use the ***returned value*** (if any), perhaps by capturing it in a variable.

4. Use ***definite loops*** and ***sequences***
   a. Write a *counted loop*, that is, a loop that iterates a given number of times, by using a *range* statement, in any of its three forms: *range(n)* *range(m, n)* *range(m, n, d)*
   b. Use the *loop variable* as called for by the problem.
   c. *Iterate through a list or string* in either of two ways, as necessary:

Looping *directly* through a list or string, e.g.

```
for item in listOfThings:
    ... item ...
```

Looping through a list or string *using its indices* as generated by a *range* expression, e.g.

```
for k in range(len(listOfThings)):
    ... listOfThings[k] ...
```

d. Use the *Accumulator Loop pattern* to accumulate things like a:

- sum
- count
- product
- list
- string
- max/min

5. Use ***conditional*** statements, in any of their 3 forms:

   `if`   `if-else`   `if-elif-elif...-else`

   - Use *comparison operators*: `==` `!=` `>` `<` `>=` `<=`
   - Use *Boolean operators*: `and` `or` `not`

6. Use ***objects***:

   a. *Construct* an object that is an instance of a *class*

   b. Apply *methods* to the object

   c. Reference *instance variables* of the object (but note: usually we use *accessor* methods instead of directly accessing the object's instance variables)

   Also, display an understanding of:

   d. How to determine what methods apply to an object

   e. The distinction between an object and a class that it is an instance of

7. ***Apply*** the above to *zellegraphics*:

   a. Construct (and hence display) a GraphWin

   b. Construct and use a Point, Line, Circle, Rectangle, Polygon, Text, Entry
   (DCM: I am not sure if Entry is fair game for Exam 1, will let you know)

   c. Apply methods to the above, including (*not all of these apply to all of the above!*):

   - *draw*
   - *undraw*
   - *move*
   - *getMouse*
   - *close*
   - *getters* like:
     - *getX*
     - *getY*
     - *getCenter*
     - *getRadius*
     - *getWidth*
     - *getHeight*
   - *setters* like:
     - *setFill*
     - *setOutline*

   d. Do an animation (using *time.sleep*)

8. ***Debug*** your code:

   a. Use Eclipse to correct *compile-time errors* like this example: `45 oops - 1 = oops`

   b. Use the *red error messages in the Console window* to know the line at which the program broke and the general nature (at least) of the error

   c. Use the *Debugger* to track down harder-to-diagnose run-time errors

9. ***Test*** your code: Supply calls in *main* or elsewhere that call your functions with parameters that help test your functions, printing returned values as appropriate.

10. ***Document*** your code, using appropriate *documentation strings* (doc-comments) and *internal comments* (with # signs)

11. ***Submit*** your code, using SVN as usual.

Sample problems of each of these items appear ***later in this document***. If you don't understand what an item here is asking, see if the example problem clarifies matters for you.

Class B items: Study these items only after you have mastered the Class A items, as they form a smaller (and less critical) part of the test.

1. ***String manipulation, input***. Be able to:
   a. Strip whitespace from the beginning and end of an input string (using ***strip***). Know when to do so.
   b. Split an input string into a list of strings (using ***split***) and then convert the strings in the list into appropriate types

2. ***String manipulation, output***. Be able to use a string's ***format*** method and associated format specifiers to do formatted output, especially lining up in columns (text or integers right-justified, floating point numbers lined up on their decimal points), limiting the precision displayed, centering and filling.

   Also, be able to print output on a single line or multiple lines, as needed.

3. ***String manipulation, other***. Be able to:
   a. Use the ***len*** function
   b. *Concatenate* (`s1 + s2`) and *duplicate* (`s * n`) strings
   c. Use *string methods* like: ***capitalize count find format index join lower replace strip split title upper***

4. ***Character manipulation***. Be able to:
   a. Use character functions *chr* and *ord*.
   b. Understand how they relate to Unicode and encoding/decoding.

5. ***List manipulation***. Be able to:
   a. Use *indexing* and *slicing*, including negative indices, in more sophisticated ways than simple indices through a sequence.
   b. Access characters inside strings inside lists, etc.
   c. Use the ***len*** function
   d. Use *list methods* like: ***append count index insert remove reverse sort***

6. ***Mutators***. Be able to:
   a. *Write mutators*.
   b. *Use mutators*.
   c. Understand how they differ from functions that return their answer.
   d. Know what data types are mutable (lists) and not mutable (strings, tuples).

7. ***Reading and writing to files***. Be able to:
    a. *Open* a text file for reading or writing. *Close* a file.
    b. *Write* to a text file using *write*
    c. *Read* from a text file using the form:

```
for line in file:
    ...
```

8. Define and use ***optional parameters***.
9. ***Return multiple values*** (as a tuple). Capture such returned values using *simultaneous assignment*.
10. Use ***type*** to determine the type of an object
11. ***Do simple procedural decomposition*** – breaking a problem into functions. (We'll do much more of this in the next few weeks. This will be a major topic of Exam 2.)

Sample Problems for the *Paper-and-Pencil* portion of Exam 1

- These sample problems are just that – *samples*. **The problems on the exam may look both *similar to* and somewhat *different from* the samples.**

> ***This page has:***
> ***Sample problems for Paper-and-Pencil #1a:***
> ***Trace short snippets of code*** (less than, say, 10 lines or so) and show ***what gets printed*** or the ***values of indicated expressions***.

1. What gets printed by the following arithmetic operations:

   a. `5 / 2`

   b. `5 // 2`

   c. `5.0 / 2 + 2`

   d. `3.0 + 1 / 2`

   e. `18 % 4`

   f. `18.0 // 4.0`

   g. `2 ** 4`

2. What gets printed by this loop?

```
for k in range(1, 4):
    print(k, k ** 2)
    print(k ** 3)
print("Done")
```

3. What gets printed by this loop?

```
for j in range(3):
    print("Hello", end='')
print("Goodbye")
```

4. What does each of the following expressions evaluate to?

   a. `list(range(2, 11, 3))`

   b. `"""xxx'hello'xxx"""`

   c. `"one" + "three"`

   d. `"one" * 3`

   e. `"101,99,-73".split(",")`

   f. `"101 99 -73".split(" ")`

   g. `"xx" + "    101 99 -73  ".strip() + "yy"`

5. Assume that the following is executed:

```
x = [10, 20, 30, 40, 50]
```

Then, what does each of the following expressions evaluate to? (If an error, indicate so.)

a. `x[2]`

b. `x[0]`

c. `x[5]`

d. `x[-1]`

e. `x[-2]`

f. `x[1:3]`

g. `x[3:1]`

h. `x[-1:-3]`

i. `x[1:5:2]`

This page has more:
Sample problems for Paper-and-Pencil #1a:
Trace short snippets of code (less than, say, 10 lines or so) and show ***what gets printed*** or the ***values of indicated expressions***.

6. Assume that the following is executed:

```
s = "forget me not"
```

Then, what does each of the following expressions evaluate to? (If an error, indicate so.)

a. `s[2]`

b. `s[0]`

c. `s[5]`

d. `s[-1]`

e. `s[-2]`

f. `s[1:3]`

g. `s[3:1]`

h. `s[-1:-3]`

i. `s[1:5:2]`

j. `s.split(" ")`

k. `"xx" + s.strip() + "yy"`

7. What gets printed by the following snippet?

```
x = [100, 20, 600, 40, 33]
for k in range(1, len(x)):
    print(k, x[k], x[k - 1 ], x[(k * 2) % 5])
```

> ***This page has more:***
> ***Sample problems for Paper-and-Pencil #1a:***
> ***Trace short snippets of code*** (less than, say, 10 lines or so) and show ***what gets printed*** or the ***values of indicated expressions***.

8. Suppose that the following functions have been defined:

```
def g(a, b):
    print("(", a, "of", b, ")")

def f(n):
    for k in range(1, n + 1):
        g(k, n)
        g(n, k)
        g(k, k)
    return n ** 2, n ** 3

def main:
    x, y = f(3)
    print(x + y)
```

Then, what does calling *main* cause to be printed? (If any error, indicate so.)

9. Suppose that the following functions have been defined:

```
def f(n):
    sum = 0
    for k in range(1, n + 1):
        sum = sum + k
    return sum

def main:
    k = 8
    sum = 10
    y = f(3)
    print(k, sum, y)
```

Then, what does calling *main* cause to be printed? (If any error, indicate so.)

> ***This page has more:***
> ***Sample problems for Paper-and-Pencil #1a:***
> ***Trace short snippets of code*** (less than, say, 10 lines or so) and show ***what gets printed*** or the ***values of indicated expressions***.

10. What gets printed by the following?

```
quotesList = ["Looks like you've been missing a lot of work lately.",
              "I wouldn't say I've been *missing* it, Bob."]
print(quotesList[1])
print(quotesList[1][2])
print(quotesList[1][2:4])
```

11. What gets printed by the following code snippet?

```
circleA = Circle(Point(100, 400), 10)
circleB = circleA
circleA.move(15, 0)
print(circleA.getCenter().getX())
print(circleB.getCenter().getX())
```

12. Suppose that the following functions have been defined:

```
def foo(x, y):
    x[0] = 100
    y = [200, 300, 400]
    y[0] = 666

def main():
    x = [1, 2, 3]
    y = [4, 5, 6]
    foo(x, y)
    print(x, y)
```

Then, what does calling *main* cause to be printed? (If any error, indicate so.)

This page has:
Sample problems for Paper-and-Pencil #1b:
Indicate errors in short snippets of code.

13. For each of the following, indicate what is wrong with it (*nothing* is a possible answer) and, if something is wrong, indicate how to correct the error.

a.
```
number1 = input("Enter an integer: ")
number2 = input("Enter an integer: ")
print(number1 * number2)
```

b.
```
number1 = 12
number2 = 5
print(average is:, (number1 + number2) / 2))
```

c.
```
number1 = 12
number2 = 5
print("average is:", (number1 + number2) // 2))
```

d.
```
s = ("jolly good!")
print("the beginning letter is:", s[1])
```

This page has:
Sample problems for Paper-and-Pencil #2a:
Write short snippets of code, especially:
***range* expressions** for common ranges

14. Write ***range*** expressions that generate:

a. `[4, 5, 6, 7, 8]`

b. `[10, 8, 6, 4, 2, 0, -2]`

c. `[0, 1, 2, 3]`

This page has:
Sample problems for Paper-and-Pencil #2b:
Write short snippets of code, especially:
loops that generate/print simple sequences

15. For each of the following, write a simple ***loop*** whose output is as given:

a.
```
4
5
6
```
etc [not the letters 'etc', but the pattern as indicated]
```
100
```

b.
```
400
397
386
```
etc [not the letters 'etc', but the pattern as indicated]
```
310
```

c. `[4, 5, 6,` *etc [not the letters 'etc', but the pattern as indicated]* `, 100]`

This page has:
Sample problems for Paper-and-Pencil #2c:
Write short snippets of code, especially:
function definitions, including those that take parameters and/or return values.

16. Write a function called power that takes two parameters, both of which should be numbers. The function returns the first parameter raised to the power specified by the second parameter.

This page has:
Sample problems for Paper-and-Pencil #2d:
Write short snippets of code, especially:
function calls, including follow-up code that uses a returned value.

4. **Explain important concepts of software development**, chosen from:
   a. The difference between a *specification* and an *implementation*, and what a *specification* of a function should include
   b. *Documentation*: how and why we put *internal comments* and *documentation strings* (*doc-comments*) in our programs
   c. *Software development tools*: what is provided by a typical, modern:
      - Integrated Development Environment (IDE)
      - Version control system
      - Debugger
   d. *Software development processes*:
      - What are some important *phases* of software development?
      - What is *procedural decomposition*, and why are functions useful?
      - What is a *compiler*? An *interpreter*?
      - What is an *algorithm*?
   e. Key ideas of *object-oriented programming*, including:
      - What makes objects different from traditional data types, namely: ***objects know stuff*** (stored in *instance variables*) and ***can do stuff*** (via *methods*)
      - Why object-oriented programming is valuable
      - The difference between a *function* and a *method*, and the different notations for invoking them
      - The difference between an *object* and a *class* to which that objects belongs
      - The difference between *accessor* and *mutator* methods
   f. What is the difference between the **`int`** and **`float`** data types? What are the limitations of each? When should use one and when the other?

While you might see some problems of type #3, don't expect a lot of such questions and don't expect them to be deep.

A simple understanding of these concepts is adequate.

The implications of the fact that variables in Python are *names* that point to *values* in memory, that is, variables are *references* to their values

1. Why do we put:

a. *Comments* in our programs?

b. *Documentation strings* in our programs?

What is the notation for each of the above?

Sample Problems for the *On-the-Computer* portion of Exam 1

- These sample problems are just that – *samples*. **The problems on the exam may look both *similar to* and somewhat *different from* the samples.**

This page has
Sample problems for On-the-Computer #1:

- ***Write short programs and/or functions*** that are examples of the ***input/compute/output pattern.***

1. Implement and test a function that prompts the user for a *radius* and a *height* and inputs values for those variables. The function then prints the volume of the cylinder with that radius and height. (Hint: $V = \pi r^2 h$)

2. Implement and test a function that prompts the user for two strings and prints the first letter of the second string immediately followed by the last letter of the first string.

This page has Sample problems for:

- ***On-the-Computer #2: Define functions*** that have ***parameters*** and (possibly) ***return values***.
- ***On-the-Computer #3: Call (invoke) functions,*** both ordinary functions and *methods*, and use the ***returned value*** (if any), perhaps by capturing it in a variable.
- ***On-the-Computer #9: Test your code.***
- ***On-the-Computer #10: Document your code.***

3. Implement and test a function called ***cylinderVolume*** that receives two parameters: a *radius* and a *height*. The function returns the volume of the cylinder with that radius and height. (Hint: $V = \pi r^2 h$)

   Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

   Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

4. Implement and test a function called ***drawPolygon*** that takes 3 parameters: a list of Points, a GraphWin and a string that represents a color. The function creates a Polygon of the given color with the given Points, and then draws that Polygon on the given GraphWin.

   Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

   Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

5. Consider the ***drawPolygon*** function in the previous exercise. You probably introduced a *local variable* in your solution to that problem. Why was a local variable helpful in the solution of that problem?

   Extra credit problem (just for grins): Implement the ***drawPolygon*** function ***without*** using a local variable.

6. Implement and test a function called ***max_min*** that takes two numbers and returns a 2-tuple: the larger of those two numbers and the smaller of those two numbers. Implementation requirement: you may *not* use the built-in ***max*** or ***min*** functions.

   Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

   Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

7. Explain, by giving a concrete example, what a *parameter* is. What a *local variable* is.

8. Write code that demonstrates that the name of a parameter in a function has nothing to do with the name of the actual argument used in a function call of that function.

This page has Sample problems for:

- ***On-the-Computer #3: Call (invoke) functions,*** both ordinary functions and *methods*, and use the ***returned value*** (if any), perhaps by capturing it in a variable.

9. Write an expression that prints the absolute value of *x*, using the built-in function *abs*.

10. Write an expression that sets the variable *z* to the sum of the numbers in list *x* and those in list *y*, using the built-in function *sum*.

11. Implement and test a function called ***use_max_min*** that has two parameters *x* and *y* and uses *max_min* (as specified in a previous problem) to print the square of the larger of *x* and *y*, followed by the cube of the smaller of *x* and *y*.

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

12. Suppose you have defined the *drawPolygon* function described in a previous sample problem. Call this function with appropriate values, constructing objects as needed.

13. Suppose that you have three Point objects (where Point is defined in *zellegraphics*), called *p1*, *p2* and *p3*. Write an expression that prints the sum of their x-coordinates.

14. Suppose that you have a Circle object (where Circle is defined in *zellegraphics*), called *circle1*. Write a statement that sets the fill color of *circle1* to **`'red'`**.

This page and the next page have Sample problems for:

- ***On-the-Computer #4: definite loops*** and ***sequences***

Also, secondarily:

- ***On-the-Computer #9: Test your code.***
- ***On-the-Computer #10: Document your code.***

15. Use ***range*** expressions to generate the following:
    - `[3, 5, 7, 9, 11]`
    - `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`
    - `[10, 9, 8, 7]`

16. Implement and test a function called ***printList*** that takes a list *w* and prints the elements of list *w*, one per line:
    - Using a *range* expression (and whatever else is necessary to answer this question)
    - Without using a *range* expression

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

17. Implement and test a function called ***accumulator*** that takes a list of numbers and returns a 4-tuple with:
    - the sum of the cubes of the numbers
    - a list containing the cubes of the numbers
    - the number in the list whose cube is largest
    - how many of the numbers are positive

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

This page and the previous page have Sample problems for:

- ***On-the-Computer #4: definite loops*** and ***sequences***

Also, secondarily: ***On-the-Computer #9: Test your code.***

On-the-Computer #10: Document your code.

18. Implement and test a function called ***backwards*** that takes a string *s* and prints *s* backwards.
    Implement this function three times:

    a. Using a relevant string function (no explicit loop).

    b. Looping directly through the string.

    c. Looping through the string *using its indices*, as generated by a *range* expression.

    Test your functions by writing code in ***main*** that calls your functions several times, each time with different parameters that help test them, printing returned values as appropriate.

    Document your functions by including appropriate *documentation strings*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

19. Implement and test a function called ***reverse*** that takes a string *s* and returns a string containing all the characters of *s* in reverse order.

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

20. Implement and test a function called ***printListOdd*** that takes a list *w* and prints the odd-numbered elements (i.e., elements 1, 3, 5, 7, etc – numbering starts at 0) of *w*, all on a single line, separated by spaces.

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

21. Implement and test a function called ***makeStringOdd*** that takes a string *s* and returns a string containing the odd-numbered elements (i.e., elements 1, 3, 5, 7, etc – numbering starts at 0) of *s*.

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

22. Implement and test a function called ***countLower*** that takes a string and returns the number of lower-case alphabetic letters in the string.

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

This page has ***Sample problems for:***

- ***On-the-Computer #5: conditionals***

Also, secondarily: ***On-the-Computer #9: Test your code.***

On-the-Computer #10: Document your code.

23. Implement and test a function called ***onlyAbc*** that takes a string and returns **`True`** if the string contains only the letters *a*, *b*, or *c* (upper or lower case, they can be repeated many times in any order, returns **`True`** in all those circumstances), else returns **`False`**.

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

24. Implement and test a function that takes a number score and prints whether the associated letter grade is an A, B, C, D or F (in the usual 10-point scale). You may assume the number score is between 0 and 100.

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

25. Implement and test a function that takes three numbers and returns the “middle” one: e.g. if the numbers are 100, 110 and 90, the returned value is 100.

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

26. Write a statement that constructs a Point object whose x-coordinate is 100 and whose y-coordinate is 50, and assigns the constructed object to a variable called p1.

27. Write a function that takes a Rectangle and returns a 2-tuple with its area and perimeter.

28. Write a statement that sets the fill color of the Circle called c to 'green'.

29. Suppose that I gave you a new module that contains methods that apply to Student objects. How would you, in Eclipse, determine what methods a Student can do?

30. Implement and test a function called ***circles*** that displays a GraphWin and causes the following to happen 5 times: when the user clicks the mouse, the program draws a circle at the point where user clicked.

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

31. Same as the previous, but when the user clicks the previously-drawn circle disappears.

32. Implement and test a function called ***lines*** that displays a GraphWin and causes the following to happen 5 times: when the user clicks the mouse twice, the program draws a line from the first mouse-click to the second one.

    Test your function by writing code in ***main*** that calls your function several times, each time with different parameters that help test it, printing returned values as appropriate.

    Document your function by including an appropriate *documentation string*. Document your function calls in ***main*** by including a BRIEF, appropriate *internal comment*.

<u>A longer problem</u>:

Write a Python program (in file *house.py*) that allows the user to provide five clicks incrementally that it uses to draw a simple five-click-house in a 600 x 500 graphics window, as depicted in the figure below. Work in stages:

a. Prompt the user to click twice in the window, upper left corner first, to specify the opposite corners of the frame of the house. The program should remember these points and use them now to draw the rectangular frame of the house.

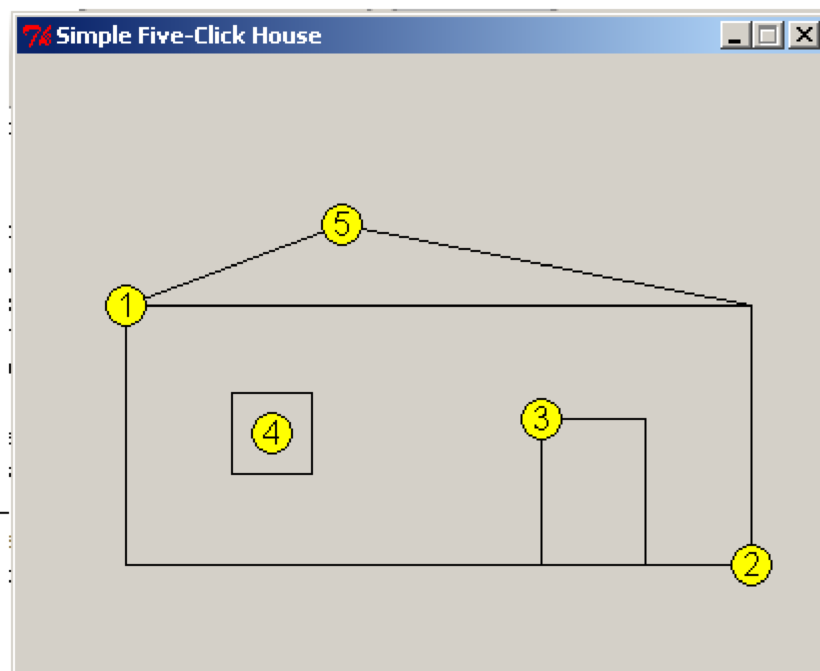
b. Prompt the user to click once on the window to specify the upper left corner of the front door. The program should draw a rectangular door that is 1/5 the width of the rectangular frame; vertically it goes from the upper left corner point to the base of the rectangular frame.

c. Prompt the user to click once to specify the center of a square window. The program should draw a square window whose side is half the height of the front door.

d. Prompt the user to click once, above the house frame, to specify the peak of the roof. The program should draw a triangular roof that extends from the point at the peak to the corners at the top edge of the house frame. [Recall zellegraphics Polygon.]

e. Animate the set of shapes so they slowly move all together off the screen.

The picture below shows where the user clicked to get the picture. Your picture should NOT include the circles at the click points.



A word of advice: This document is *huge*, because it tries to indicate *everything* that you might be expected to do on Exam 1. Of course, you won't be tested on all of this!

Prioritize your efforts:

1. Skim this page, so that you know the format of the exam and what resources you are permitted for each of the 2 parts.
2. Skim pages 2-3 (for Paper-and-Pencil part) and 4-5 (for On-the-Computer part). For each, circle the items that you are unsure about.
3. Talk to someone – your instructor, Review Session people, classmates, trusted friends – about your circled items. Make notes as needed.
4. As time permits, choose some Practice Problems to try – ones that you *don't* know how to do (skip the ones that you *do* know how to do) and that you think might help your learning and your score on the test. Strive for big ideas first.
5. As time permits, either solidify your understanding of the big-ticket ideas, or go back to pages 6-7 for lesser items on the On-the-Computer part and repeat step 3 on those. Try more Practice Problems, either big-ticket ideas of the lesser ideas, depending on where you are in your mastery of this course.

Finally, note that we have covered at least 75% of the ideas of the entire course already. Over the next 7 weeks, we do the critical step of practice, practice, practice (especially in the form of a way-cool project, followed by seeing the same Python ideas but in