# Lecture 12 - Exercises

## 12B - Tuples

1. What does the following code print?

```
a = 1
b = 2
c = 3

v = (a, a, c)

print(v, sep=" ")
```

2. What does the following code print?

```
a = 1
b = 2
c = 3

a, b, c = (a, a, c)

print(a, b, c, sep=" ")
```

3. Suppose the following code is executed. The numbers at left are line numbers and are not part of the code itself. If any line causes an error, assume that line is not run and continue execution with the next line.

```
1   a, b, c = 6, 4, 2
2   (z, x) = c, b
3   print((x, z))
4   v = (x, z, c)
5   print(v)
6   a, b = v
```

    1. Which line(s), if any, cause errors? List all that apply.
    2. What does line 5 print?

4. Consider the following function:

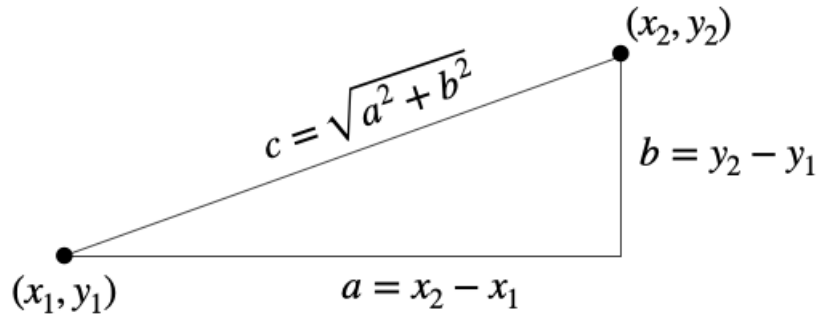
```
def f(a, b):
  xa, ya = a
  xb, yb = b
  return (ya, xa), (yb, xb)
```

   For each of the following calls to the above function, say what the function returns, or “error” if an error will occur.

    1. `f(1, 2)`
    2. `f(1, 2, 3, 4)`
    3. `f((1, 2), (3, 4))`
    4. `f(((1, 1), 2), (3, 4))`
    5. `f((1, 1, 2), (3, 4))`

## Problems

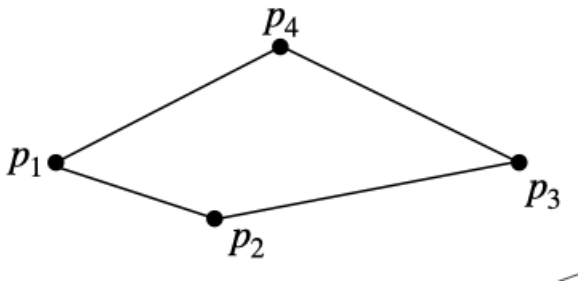
1. Recall that the distance between two points can be calculated using the pythagorean theorem. The distnace between the two points is the hypotenuse of a triangle formed by the difference in $x$ coordinates and the difference in $y$ coordinates:




With this in mind, implement the following function:

```
def distance(p1, p2):
    """ Return the distance between 2D points p1 and p2.
    Precondition: p1 and p2 are 2-tuples containing the
    x and y coordinates of the two points. """

    # your code here
```

2. A common task in computer graphics is to *triangulate* a polygon - or in other words, generate a collection of triangles that cover a polygon. A simple case of this is triangulating a quadrilateral - or in other words, splitting a 4-sided shape into two triangles. We'll assume that the quadrilateral in question is convex (i.e., the line between any pair of vertices does not leave the quadrilateral). We can triangulate this shape by connecting one pair of opposite points with a line. For example, the following quadrilateral can be triangulated by either connecting $p_1$ and $p_3$ or connecting $p_2$ and $p_4$. For various reasons it turns out to be desirable to have triangles whose smallest angle isn't too small; this means that in the shape below, a better triangulation would be to connect $p_2$ and $p_4$.




Implement the following function, which returns the pair of opposite points that are closest together. You can make use of the `distance` function you implemented above.

```
def best_divider(p1, p2, p3, p4):
    """ Return the closest pair of opposite points on the quadrilateral formed by p1,
    p2, p3, and p4. Precondition: p1-p4 are 2-tuples of numbers specifying a
    quadrilateral in counter-clockwise order. """
```

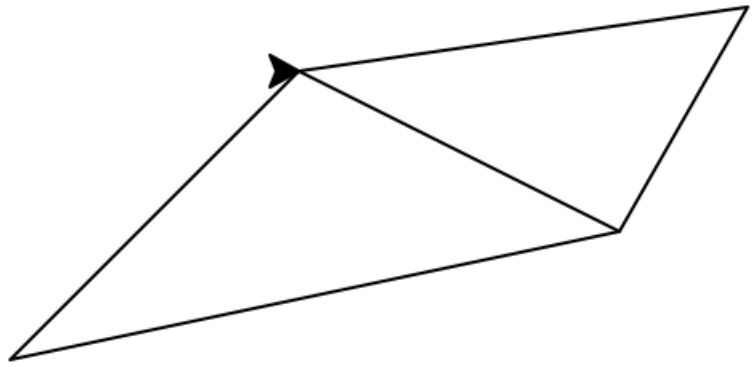
3. Implement the following function to draw a triangulation of a quadrilateral using a turtle. You can make use of any of the functions you've written so far. You also may want to check out the turtle's `goto` method to help make things a little simpler.

```
def draw_triangulation(t, p1, p2, p3, p4):
    """ Draw the triangulated quadrilateral defined by vertices p1, p2, p3, p4 using
    the turtle t. The triangulation connects the closer pair of opposing points.
    Precondition: t is a turtle; p1-p4 are 2-tuples of numbers specifying a
    quadrilateral in counter-clockwise order."""
```

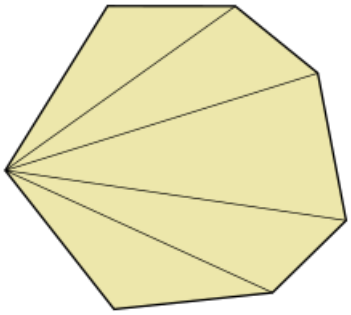
Test your program using a main program like the following:

```
import turtle

# define your function(s) here

q = turtle.Turtle()
p1 = (10, 10)
p2 = (200, 50)
p3 = (240, 120)
p4 = (100, 100)
draw_triangulation(q, p1, p2, p3, p4)
```

When I run this, I get a drawing in the turtle window that looks like this:



4. Polygons with more vertices are trickier to optimally triangulate, because there are more choices of pairs of edges to connect. A simple, foolproof method for getting *a* triangulation (but not necessarily an *optimal* one) is called a *fan trianguation*. Starting at one vertex, a line is drawn to connect that vertex to each other one that's not adjacent to it, as in the example image below:



Implement the following function, which draws a fan-triangulated picture of a polygon specified by a *list* of vertices.

```
def draw_fan_triangulation(t, poly):
    """ Draw a fan-triangulated picture of the polygon specified by poly.
    Preconditions: t is a turtle; poly is a list of 3 or more 2-tuples of numbers
    that give the vertices of a convex polygon in counter-clockwise order. """
```

*Hint:* You may find it helpful to be able to access a specific element of the list: you can do this with `poly[i]`, where `i` is the *index* of the vertex you want, starting with 0 as the first element.

Write a main program to test your function.

5. See if you can divise a method for triangulating a convex polygon that will minimize long-and-narrow triangles. This could involve a fan triangulation with a carefully chosen starting vertex, or a different approach entirely. Focus on the algorithm first, writing it in pseudocode; once you've got something you think will work, feel free to go ahead and write Python code to implement it.