

Computer Programming 11

Unit 4: Graphics and Animation


RELATED CONTENT

 [History – Turtle Graphics](#)

 [Python Turtle Commands](#)

 [Turtle Color List](#)

PYTHON CODE CONCEPT

 [Our First Turtle Program](#)

4.1 Python Turtle Module

An easy way to work with graphics in Python is to use the Turtle Module. Turtle graphics have been around for a long time and were used as part of the Logo programming language to help kids learn to code.

If you look in the Lib folder of your Python installation directory you will see a module called turtle.py. This module was created by Gregor Lingl as a nod to the old Logo coding language taught in the 80's in schools.

turtle_stamp.py – Practice (2 marks)

We are going to test out some of the turtle commands in Python.

Create your **turtle_stamp.py** file and save it to your network (A: drive) folder.

Type the header. We won't worry about an algorithm for this simple practice assignment.

```
1 # Drawing Turtle Does What I Say
2 # Author: Your Name
3 # Date: Current Date
4
5 import turtle
6
7 # Create a turtle object
8 bucky = turtle.Turtle()
9
10 # Move forward 50 pixels and stamp
11 bucky.forward(50)
12 bucky.stamp()
13
14 # Move forward 50 pixels and stamp
15 bucky.forward(50)
16 bucky.stamp()
```

In the code above, **bucky** is an "object" of the class type Turtle. Meaning **bucky** can be asked to do all the things a Python Turtle can do.

Now let's get our turtle to repeat drawing a line and placing a stamp. Change your code so it looks like this:

```

1 # Drawing Turtle Does What I Say
2 # Author: Your Name
3 # Date: Current Date
4
5 import turtle
6
7 # Create a turtle object
8 bucky = turtle.Turtle()
9
10 # Repeat 10 times
11 for i in range(10):
12     # Move forward 20 pixels and stamp
13     bucky.forward(20)
14     bucky.stamp()
15

```

You should see your turtle, named bucky, travel across the screen and stamp 10 times.

Now let's make our turtle interactive by asking the user for input and having our turtle perform their action.

```

1 # Drawing Turtle Does What I Say
2 # Author: Your Name
3 # Date: Current Date
4
5 import turtle
6
7 # Create a turtle object
8 bucky = turtle.Turtle()
9 # Change the shape to an actual turtle
10 bucky.shape("turtle") #other options: arrow, circle, square, triangle, classic
11 bucky.color("green")
12
13 # Get user command
14 command = input("What is your command: forward(f), stamp(s) \n> ")
15
16 # if the user types f, go forward 20
17 if command == "f":
18     bucky.forward(20)
19 elif command == "s":
20     bucky.stamp()
21

```

Of course, our turtle won't get very far. We should let the user keep drawing by adding a while loop. It would be also helpful if our turtle could turn so it doesn't draw off the screen.

When our user wants to stop drawing, we can break out of the loop if they type the "sentinel" or "flag" that we've designated in the instructions to them.

Change your code so it looks like the following example and note we've added a drawing surface called paper:

```

1 # Drawing Turtle Does What I Say
2 # Author: Your Name
3 # Date: Current Date
4
5 import turtle
6
7 # Create a screen that we can close
8 paper = turtle.Screen()
9 paper.bgcolor("lightyellow")
10
11 # Create a turtle object
12 bucky = turtle.Turtle()
13 # Change the shape to an actual turtle
14 bucky.shape("turtle") #other options: arrow, circle, square, triangle, classic
15 bucky.color("green")
16

```

PYTHON CODE CONCEPT

🔗 While Loop and Sentinel Values

```

16
17 while True:
18     # Instructions
19     print("You can ask the turtle to...")
20     print("Move forward (f)")
21     print("Turn right (r)")
22     print("Stamp (s)")
23     print("Stop (stop)")
24     # Get user command and convert to lower case
25     command = input("What is your command: \n> ").lower()
26
27     # if the user types f, go forward 20
28     if command == "stop":
29         break
30     elif command == "f":
31         bucky.forward(20)
32     elif command == "r":
33         bucky.right(90)
34     elif command == "s":
35         bucky.stamp()
36
37 # exits properly
38 paper.exitonclick()

```

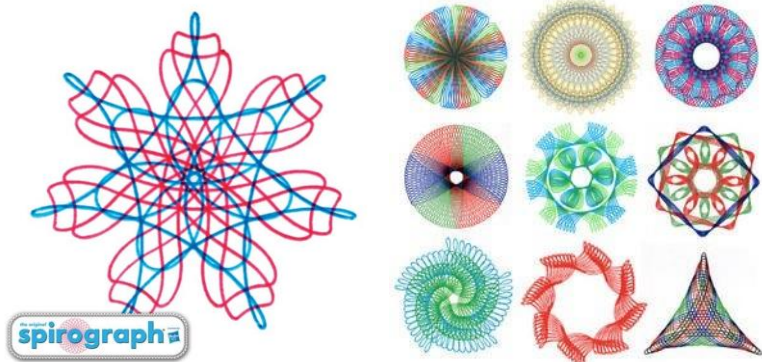
The while loop has a break which is triggered by the "stop" sentinel value being entered by the user.

Also, now that we've added a `.Screen()` and `.exitonclick()` the program will close without hanging after the user is done drawing.

As a last step, add more commands for turning left and try changing the turtles pen size `bucky.pensize(3)` and colors.

BONUS:

Could you get your turtle to draw a cool pattern like these?

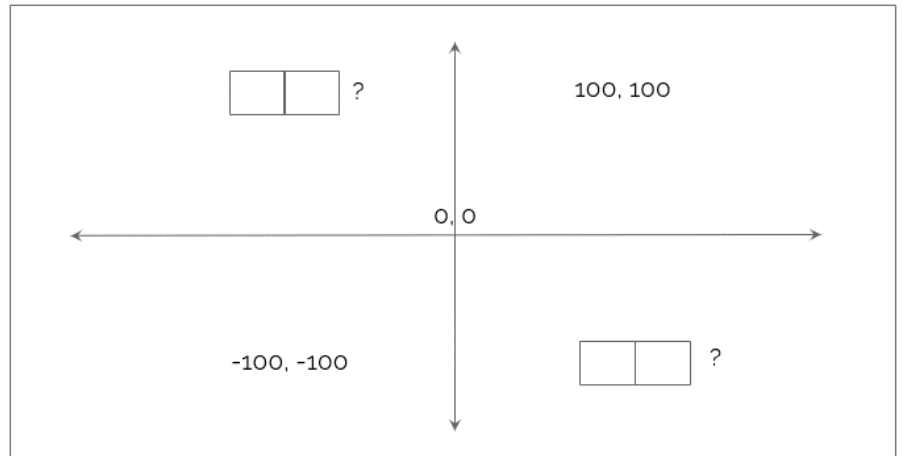


4.2 Coordinates & Using Functions

For graphics we work with a horizontal (x) and vertical (y) coordinate system.

Coordinate systems on the computer can vary between applications. Typically we view the top left corner of the screen as having an x,y value of 0,0. The bottom right corner of the screen depends on your viewing resolution. Mine is currently 1920, 1080.

Turtle graphics coordinates work a little differently. 0,0 is the center of the window drawing surface.



For example, if your drawing surface had a size of 200 pixels x 200 pixels the coordinate system would be as displayed above for turtle graphics.

cookies_with_functions.py – Practice (2 marks)

We are going to get our turtle to draw a cookie for us and then get it to draw multiple cookies using a function.

Create your **cookies_with_functions.py** file and save it to your network (A: drive) folder.

Enter the following code:

```
1  # Cookies with Functions
2  # Author: Your Name
3  # Date: Current Date
4
5  import turtle
6
7  # Create a drawing surface
8  drawing_surface = turtle.Screen()
9
10 # Set position and size of drawing surface's window
11 drawing_surface.setup(500, 500, 20, 20)
12 drawing_surface.bgcolor("lightblue")
13
14 # Create a turtle object
15 angela_turtlesbury = turtle.Turtle()
16
17 # Set the shape and pen color
18 angela_turtlesbury.shape("classic")
19 angela_turtlesbury.color("brown")
20
21 # exits properly
22 drawing_surface.exitonclick()
```

Run it and make sure it is working correctly. You should get a light blue screen near the top left corner of your monitor with a brown triangle in the middle.

If it worked, try changing some of the .setup numbers to get a feel for what they do.

If you haven't already, reset the .setup numbers to match the original code. Then add the following code to draw the outside diameter of the cookie.

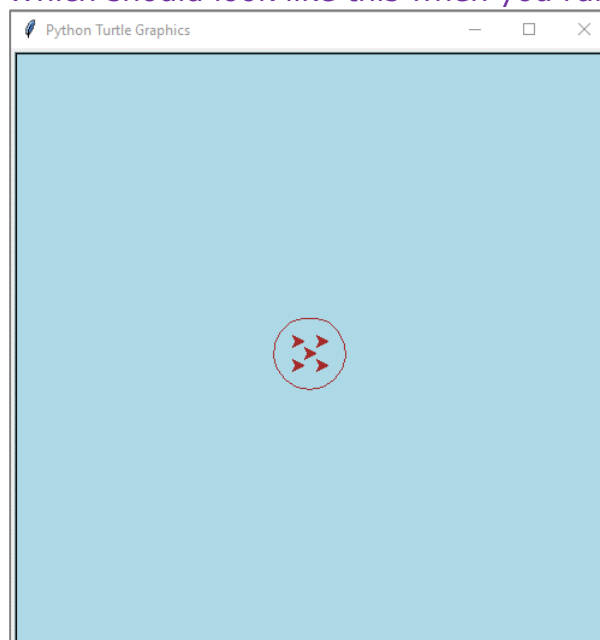
```
20
21 # Draw a cookie at an x,y location
22 angela_turtlesbury.penup() # lift pen off drawing surface
23 angela_turtlesbury.goto(-5, -30) # set start point
24 angela_turtlesbury.pendown() # put pen on drawing surface
25 angela_turtlesbury.circle(30) # draw a circle with 30 pixel diameter
26 angela_turtlesbury.penup()
27
28 # exits properly
29 drawing_surface.exitonclick()
```

Test the code. You should see a brown circle near the center of the screen.

Now add the chips.

```
27
28 # Chocolate chip in the middle
29 angela_turtlesbury.goto(0, 0)
30 angela_turtlesbury.stamp()
31
32 # Chocolate chip top left
33 angela_turtlesbury.goto(-10, 10)
34 angela_turtlesbury.stamp()
35
36 # Chocolate chip bottom left
37 angela_turtlesbury.goto(-10, -10)
38 angela_turtlesbury.stamp()
39
40 # Chocolate chip bottom right
41 angela_turtlesbury.goto(10, -10)
42 angela_turtlesbury.stamp()
43
44 # Chocolate chip top right
45 angela_turtlesbury.goto(10, 10)
46 angela_turtlesbury.stamp()
47
48 # exits properly
49 drawing_surface.exitonclick()
```

Which should look like this when you run it.



PYTHON CODE CONCEPT

🔗 [Intro to Functions](#)

🔗 [Functions are Recipes!](#)

🔗 [Function Example: Turtle Bar Chart](#)

Now, what if I want multiple cookies? If I use a **for** loop it would just draw the cookie in the same spot.

We want to be able to change the location of the cookies on our drawing surface.

Functions are used to reduce repetitive code and perform specific tasks multiple times. Please visit the links listed to the left to learn more about how functions work in Python.

You've already been using built-in functions. For example:

```
bucky.forward(50)
angela_turtlesbury.stamp()
```

.forward() is a function in the turtle module and we sent it an argument of **50** so that the function would move the turtle forward by the parameter of **50**.

.stamp() is a function in the turtle module that doesn't need an argument sent to it to do its job.

You can make your own functions in Python too!

- 1.) Use the keyword **def**
- 2.) Choose a **name** for the function. Use good convention/style. Names should be descriptive and all lowercase with underscores.
- 3.) **Indent** the code that is inside the function.
- 4.) Leave 2 blank lines after a function.
- 5.) Functions should also have documentation called a **docstring**. The style is to use `""" text """` right below the function header. The docstring describes what the function does, what input it needs for any parameters, and what output the function will produce if it has a **return** statement.
- 6.) Functions are placed at the top of the program code below import statements and constants or global variables.

We will use a function to draw the cookie (below). Note the name of the function and that the content of the function is indented. There is a docstring that describes what the function does. We leave 2 blank lines after a function.

```

17 def draw_cookie():
18     """ Draws a cookie """
19
20     # Set the shape and pen color
21     angela_turtlesbury.shape("classic")
22     angela_turtlesbury.color("brown")
23
24     # Draw a cookie at an x,y location
25     angela_turtlesbury.penup() # lift pen off drawing surface
26     angela_turtlesbury.goto(-5, -30) # set start point
27     angela_turtlesbury.pendown() # put pen on drawing surface
28     angela_turtlesbury.circle(30) # draw a circle with 30 pixel diameter
29     angela_turtlesbury.penup()
30
31     # Chocolate chip in the middle
32     angela_turtlesbury.goto(0, 0)
33     angela_turtlesbury.stamp()
34
35     # Chocolate chip top left
36     angela_turtlesbury.goto(-10, 10)
37     angela_turtlesbury.stamp()
38
39     # Chocolate chip bottom left
40     angela_turtlesbury.goto(-10, -10)
41     angela_turtlesbury.stamp()
42
43     # Chocolate chip bottom right
44     angela_turtlesbury.goto(10, -10)
45     angela_turtlesbury.stamp()
46
47     # Chocolate chip top right
48     angela_turtlesbury.goto(10, 10)
49     angela_turtlesbury.stamp()
50
51     # call the draw_cookie function
52     draw_cookie()
53
54     # exits properly
55     drawing_surface.exitonclick()
56

```

Our function is still going to draw cookies in the same spot. So, we need to add some parameters.

We add **parameters** to the function declaration inside brackets and using commas to separate each **parameter**.

Then we use the **parameters** in the code of the function. They are essentially variables that are local to the function and can only be used inside the function.

When we **call the function**, we send **arguments** that become the **values** inside the **parameters**.

IMPORTANT: The order of the arguments and the data type of the arguments must be the same as the parameters of the function.

Change your code so it looks like the following.

```

1  # Cookies with Functions
2  # Author: Your Name
3  # Date: Current Date
4
5  import turtle
6

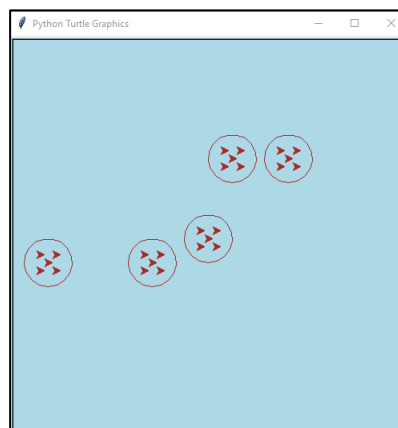
```

```

7  def draw_cookie(x,y):
8      """ Draws a cookie at a specified coordinate.  Receives
9          two coordinates """
10
11     # Set the shape and pen color
12     angela_turtlesbury.shape("classic")
13     angela_turtlesbury.color("brown")
14
15     # Draw a cookie at an x,y location
16     angela_turtlesbury.penup() # lift pen off drawing surface
17     angela_turtlesbury.goto(-5 + x, -30 + y) # set start point
18     angela_turtlesbury.pendown() # put pen on drawing surface
19     angela_turtlesbury.circle(30) # draw a circle with 30 pixel diameter
20     angela_turtlesbury.penup()
21
22     # Chocolate chip in the middle
23     angela_turtlesbury.goto(0 + x, 0 + y)
24     angela_turtlesbury.stamp()
25
26     # Chocolate chip top left
27     angela_turtlesbury.goto(-10 + x, 10 + y)
28     angela_turtlesbury.stamp()
29
30     # Chocolate chip bottom left
31     angela_turtlesbury.goto(-10 + x, -10 + y)
32     angela_turtlesbury.stamp()
33
34     # Chocolate chip bottom right
35     angela_turtlesbury.goto(10 + x, -10 + y)
36     angela_turtlesbury.stamp()
37
38     # Chocolate chip top right
39     angela_turtlesbury.goto(10 + x, 10 + y)
40     angela_turtlesbury.stamp()
41
42
43     # Create a drawing surface
44     drawing_surface = turtle.Screen()
45
46     # Set position and size of drawing surface's window
47     drawing_surface.setup(500, 500, 20, 20)
48     drawing_surface.bgcolor("lightblue")
49
50     # Create a turtle object
51     angela_turtlesbury = turtle.Turtle()
52
53     # call the draw_cookie function
54     draw_cookie(0,0)
55     draw_cookie(100,100)
56     draw_cookie(-70,-30)
57     draw_cookie(30,100)
58     draw_cookie(-200,-30)
59
60     # exits properly
61     drawing_surface.exitonclick()

```

Try using some different arguments. Have your program finish by drawing 5 cookies on the screen.



4.3 Passing Data between Functions

Functions are called by stating them in the code and passing arguments inside brackets if the function has parameters that need to be fulfilled.

A function can call another function and pass along arguments.

```
1 import turtle
2
3 def instructions():
4     """ Inform the user about this program, get user's color choice,
5     send users color choice to the draw_circle function """
6
7     # instructions
8     print("Hello, I will draw a circle.")
9     print("What color would you like it to be?")
10
11     # get color
12     prompt = ("Red, Green, Blue \n>")
13     color=input(prompt).lower()
14     while color != 'red' and color != 'green' and color != 'blue':
15         color=input(prompt).lower()
16
17     # draw circle with the user's color
18     draw_circle(color)
19
20
21 def draw_circle(circle_color):
22     """ Draws a circle with received color """
23     # setup the window
24     drawing_surface = turtle.Screen()
25
26     # Create a turtle object
27     sue = turtle.Turtle()
28
29     # Set the pen color
30     sue.color(circle_color)
31
32     # Draw a circle
33     sue.circle(80) # draw a circle with 30 pixel diameter
34
35     # exits properly
36     drawing_surface.exitonclick()
37
38
39 instructions()
```

In the example above the yellow arrows show that the first function call when the program runs is line 39 **instructions()** which calls the **instructions()** function on line 3.

On line 18 the **instructions()** function is calling the **draw_circle()** function and sending **color** as the argument. The function header on line 21 contains the **circle_color** parameter which will receive the value from the **color** argument.

Functions should contain docstrings and have 2 blank lines at the end. Note the green arrows and 2's above.

interactive_drawing.py – Exercise (5 Marks)

You are going to create an application that gives instructions to a person and asks them to make some choices about what a drawing should look like.

The drawing your application makes should be interesting and unique. It should not be the same as a classmate's program. While you can share ideas, your code must be your own and it must be unique.

Create a file called **interactive_drawing.py** in your network folder on (A:) drive.

Your code must include the following:

- Good Style
 - algorithm
 - naming conventions
 - blank line spacing – 2 after each function, 1 between programming blocks
 - programmer commenting
 - functions at the top immediately after import statements
 - program code that isn't inside a function should be written after the functions have been declared
- Information for the user – description of the app and clear instructions for what the app will draw and what the user's input choices are
- At least two functions. Functions should have one or more parameters.
- The first function calls the next function by passing arguments along.
- Nicely formatted and descriptive output
- A drawing surface and exit statement so that your program does not crash when closing
- Your program must be different from classmates


Note: You can have more than 2 functions if you wish but at least 2 functions must match the criteria listed above.

Watch the video of examples to the left to get an idea of what is expected for output.

Evaluation (5 Marks)

- /1 Good Style
- /1 Minimum of 2 Functions have docstring
- /1 A function calls another sending arguments to parameters
- /1 Program closes properly using drawing surface and exitonclick
- /1 Program is creative and unique and utilizes input from a human as part of the drawing

VIDEO TUTORIAL

 [Examples of Functions calling Functions](#)

PYTHON CODE CONCEPT

🔗 [Visualizing Recursion](#)

RELATED CONTENT

🔗 [When to use Recursion](#)

Review Questions:

1.) What would this code output?

```
count = 10
while count > 0:
    print(count)
    count = count - 1
```

2.) What would this draw?

```
import turtle
alex = turtle.Turtle()

for i in range(4):
    alex.forward(50)
    alex.left(90)
```

3.) What would this code output?

```
seed = ""
while len(seed) < 10:
    seed = seed + "o"
print(seed)
```

4.4 Function Recursion

A recursive function is a function that calls itself. It is another way to reduce repetitive code.

Recursion is used when iteration with for/while can't provide enough complexity to solve the problem. Usually this is when multiple branching paths need to be created or searched.

A recursive function has two parts. The **recursive call** and the **base case**.

```
def count_down(number):
    """ Counts backwards from the starting number.
    Recursive until 0 is reached. """
    print(number)
    if (number == 0): # base case
        print("Blast Off!")
    else:
        number -= 1
        count_down(number) # recursive call

user_num = int(input("Please enter a number. \n>"))
count_down(user_num)
```

In the example above, the **base case** is the condition that checks to see if we have gotten the information we need from the function. If the **base case** is achieved, then the function will stop calling itself.

The **recursive call** will keep calling the function from inside of itself until the base case is met.

It can be complex and challenging to visualize so we will be using a fractal tree to assist in understanding recursion.

recursive_tree.py – Practice (2 marks)

We are going to get our turtle to draw a tree using recursion to demonstrate the complex branching that recursive functions can create.

Create your **recursive_tree.py** file and save it to your network (A: drive) folder.

Enter the following code:

```
1  # Recursive Tree Challenge
2  # Author: Your Name
3  # Date: Current Date
4
5  # This program uses recursion to draw a tree.
6
7  import turtle
8
9  def draw_tree(level, branch_length):
10     """ A recursive function to draw trees
11         level - the number of levels of branches
12         branchlength - the length of the branch to draw
13     """
14
15     # NOT AT the leaf level yet
16     if level > 0:
17         # 1 - Draw a branch
18         turtle.forward(branch_length)
19         # 2 - Turn left and make a mini tree
20         turtle.left(40)
21         draw_tree(level-1, branch_length/1.61) # RECURSIVE CALL!
22         # 3 - turn right and make a mini tree
23         turtle.right(80)
24         draw_tree(level-1, branch_length/1.61) # RECURSIVE CALL!
25         # 4 - go back
26         turtle.left(40)
27         turtle.back(branch_length)
28     # leaf level reached (level == 0)
29     else: # BASE CASE!
30         # Stamp the leaf
31         turtle.color("green")
32         turtle.stamp()
33         turtle.color("brown")
34
35
36     #----- Main Program Starts Here -----#
37
38     # Create a drawing surface
39     drawing_surface = turtle.Screen()
40
41     # Move the turtle
42     turtle.speed(0)
43     turtle.penup()
44     turtle.goto(0,-180)
45     turtle.left(90)
46     turtle.pendown()
47
48     # Setup drawing
49     turtle.color("brown")
50     turtle.width(3)
51     turtle.shape("triangle")
```

```
52
53 # Draw a tree using a recursive function
54 draw_tree(1, 120)
55
56 # exits properly
57 drawing_surface.exitonclick()
```

Test yours to see if it's working. If you did it correctly you will see only 1 branch 😊



Change line 54 so it reads as follows:

```
53 # Draw a tree using a recursive function
54 draw_tree(2, 120)
```

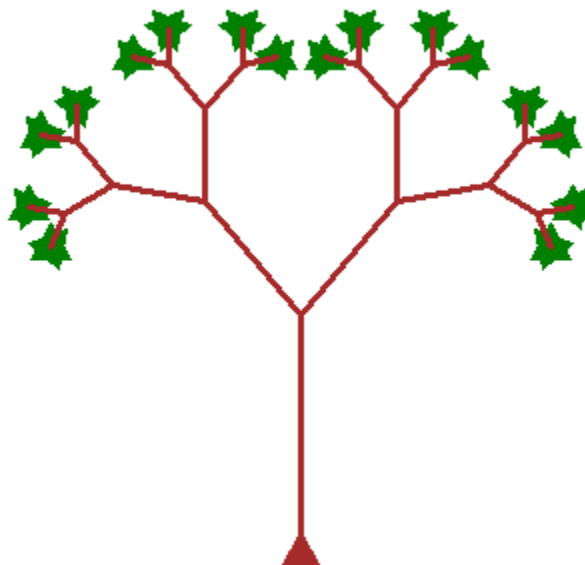
When you run your program, you should now see this:



Change line 54 again so it reads as follows:

```
53 # Draw a tree using a recursive function
54 draw_tree(5, 120)
```

You should now see this:



To complete this assignment your challenge is to modify your recursion so that the tree looks like this:



Notice that there is a change in the shape of the fractal branches. Have fun! 😊

When you have finished this final challenge, and your assignment is complete, please make sure it is saved in your network folder (A: drive) and show your teacher for marks.

4.5 Dictionaries

So far, we've looked at strings and lists which are considered sequential collections of data. We use index values to find data at a location within the string or list.

Dictionaries are a different type of collection that is unordered and does not utilize index values. Instead information is organized using **key-value pairs**.

Example:

```
dct_member_info = {"Name": "Bob Smith", "Age": 36, "Location": "Arizona"}
```

Notice the dictionary `dct_member_info` has content inside **curly brackets**.

The **keys** are `Name`, `Age`, and `Location`

The **values** are `"Bob Smith"`, `36`, and `"Arizona"`.

The **key** in a dictionary is considered immutable – meaning it cannot be changed. Keys inside a dictionary must be uniquely named – no two keys can have the same name.

PYTHON CODE CONCEPT

🔗 [Dictionaries](#)

🔗 [Dictionaries 101: Detailed Visual Intro](#)

The **values**, associated with each key in a dictionary, can be changed.

We use the key in the dictionary to get the value. In this example:

```
dct_member_info = {"Name": "Bob Smith", "Age": 36, "Location": "Arizona"}
print(dct_member_info["Name"])
```

Generates this output:

```
Bob Smith
```

dictionary_tree.py – Practice (2 marks)

Open your **recursive_tree.py** file from the previous assignment and save it with the new name **dictionary_tree.py**.

It is still going to be recursive but now we are going to get our turtle to draw a tree using different leaf color values pulled from a dictionary.

Add the color dictionary (see lines 15-17 below) Don't worry if your recursive branching is slightly different from the one below. The dictionary uses hexadecimal colors (see Related Content Link to the left)

RELATED CONTENT

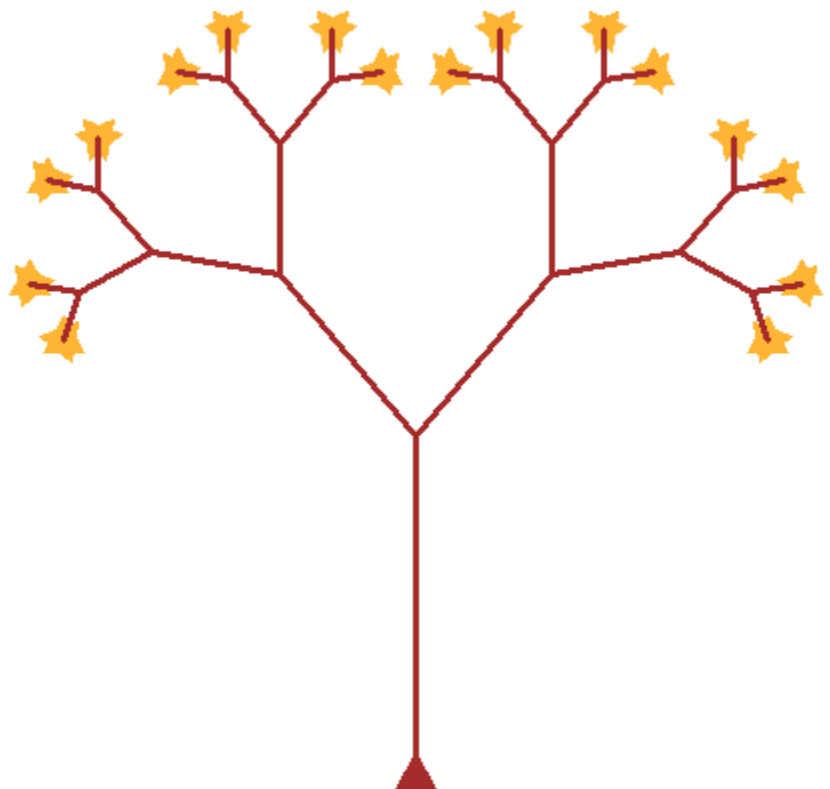
 [Hexadecimal Color Code picker for HTML](#)

```
1  # Recursive Tree With Dictionary
2  # Author: Your Name
3  # Date: Current Date
4
5  # This program uses recursion to draw a tree.
6
7  import turtle
8
9  def draw_tree(level, branch_length):
10     """ A recursive function to draw trees
11         level - the number of levels of branches
12         branchlength - the length of the branch to draw
13     """
14
15     # Dictionary containing colours for seasons
16     seasonal_colour = {"autumn": "#FFB533", "spring": "#FCB3F5",
17                       "summer": "#12C517", "winter": "#BFF5F5"}
18
19     # NOT AT the leaf level yet
20     if level > 0:
21         # 1 - Draw a branch
22         turtle.forward(branch_length)
23         # 2 - Turn left and make a mini tree
24         turtle.left(40)
25         draw_tree(level-1, branch_length/1.61) # RECURSIVE CALL!
26         # 3 - turn right and make a mini tree
27         turtle.right(80)
28         draw_tree(level-1, branch_length/1.61) # RECURSIVE CALL!
29         # 4 - go back
30         turtle.left(40)
31         turtle.back(branch_length)
32     # leaf level reached (level == 0)
33     else: # BASE CASE!
34         # Stamp the leaf
35         turtle.color(seasonal_colour["autumn"])
36         turtle.stamp()
37         turtle.color("brown")
38
```

Then change the leaf color to the value of a key from the dictionary (see line 35)

```
39  
40 #----- Main Program Starts Here -----#  
41  
42 # Create a drawing surface  
43 drawing_surface = turtle.Screen()  
44  
45 # Move the turtle  
46 turtle.speed(0)  
47 turtle.penup()  
48 turtle.goto(0,-180)  
49 turtle.left(90)  
50 turtle.pendown()  
51  
52 # Setup drawing  
53 turtle.color("brown")  
54 turtle.width(3)  
55 turtle.shape("triangle")  
56  
57 # Draw a tree using a recursive function  
58 draw_tree(5, 170)  
59  
60 # exits properly  
61 drawing_surface.exitonclick()
```

Your tree should now have a different leaf colour.



user_dictionary_tree.py – Practice (2 marks)

Open your **dictionary_tree.py** file from the previous assignment and save it with the new name **user_dictionary_tree.py**.

We are now going to make the dictionary tree customizable so that a person running the program could choose the color, number of branches, and size of the tree.

The user's input will be collected as values for a second dictionary.

Begin by adding a **get_user_tree()** function to your code after the import statement and before the **draw_tree()** function:

```
8 import turtle
9
10 def get_user_tree():
11     """ Builds a dictionary of values by getting input from the user. Calls the
12         function to draw the tree and passes the dictionary as an argument """
13
14     # Dictionary containing colours for seasons
15     seasonal_colour = {"autumn": "#FFB533", "spring": "#FCB3F5",
16                       "summer": "#12C517", "winter": "#8FF5F5"}
17
18     # initialize user's tree dictionary
19     user_choices = {"Leaves": "", "Branches": 0, "Size": 0}
20
21     # get leaf color and add it to the dictionary
22     while user_choices["Leaves"] == "":
23         print("What season is it?")
24         leaf_color = (input("autumn, spring, summer, winter? \n>")).lower()
25         if leaf_color in seasonal_colour:
26             user_choices["Leaves"] = seasonal_colour[leaf_color]
27         else:
28             user_choices["Leaves"] = ""
29
30     # get number of branches and add it to the dictionary
31     while user_choices["Branches"] == 0 or user_choices["Branches"] > 8:
32         branch_number = int(input("How many branches (1 to 8)? \n>"))
33         user_choices["Branches"] = branch_number
34
35     # get size of initial branch and add it to the dictionary
36     while user_choices["Size"] < 50 or user_choices["Size"] > 240:
37         tree_size = int(input("How big (50-240)? \n>"))
38         user_choices["Size"] = tree_size
39
40     # Draw the tree using the user's choices dictionary
41     draw_tree(user_choices)
42
43
```

Note: The **seasonal_colour** dictionary has been moved into the **get_user_tree()** function and should be removed from the **draw_tree()** function.

The **get_user_tree()** function will build a dictionary of user responses. On line 19 you can see the new dictionary. It must be initialized before values can be added by the user.

The **while** statements ensure that user input is appropriate.

The function call in line 41, `draw_tree(user_choices)` is now sending the dictionary of user values as an argument to the `draw_tree()` function.

The `draw_tree(level, branch_length)` function had two parameters for two integers. It needs to be changed so that it has one parameter for a dictionary instead.

```
44 def draw_tree(dct_tree_specs):
45     """ A recursive function to draw trees
46     dct_tree_specs = a dictionary with color, number of branches, and size
47     """
48
```

Make sure you also update your function documentation to reflect the change.

When the dictionary that we are passing, is called recursively from the `draw_tree(dct_tree_specs)` function it must get smaller. This presents an extra challenge where dictionaries are concerned. We can't just subtract or divide the values as we would a variable because that would permanently change the values inside the dictionary.

So, we will need to create a new `dct_tree_smaller` dictionary to hold the color and the reduced values for number of branches and size.

Please add this dictionary inside your `draw_tree(dct_tree_specs)` function.

```
49     # initialize new dictionary to hold users seasonal color and reducing
50     # branches and branch length
51     dct_tree_smaller = {"Leaves": dct_tree_specs["Leaves"],
52                        "Branches": 0,
53                        "Size": 0}
54
```

The remainder of the code for the function is similar to what it was before. However, instead of changing values for variables we work with the two dictionaries of user values. The original (specs) and the reduction for recursion (smaller)

```
55     # NOT AT the leaf level yet
56     if dct_tree_specs["Branches"] > 0: # level > 0
57         # 1 - Draw a branch
58         turtle.forward(dct_tree_specs["Size"])
59         # 2 - Turn left and make a mini tree
60         turtle.left(40)
61         dct_tree_smaller["Branches"] = dct_tree_specs["Branches"]-1
62         dct_tree_smaller["Size"] = dct_tree_specs["Size"]/1.61
63         draw_tree(dct_tree_smaller) # RECURSIVE CALL!
64         # 3 - turn right and make a mini tree
65         turtle.right(80)
66         dct_tree_smaller["Branches"] = dct_tree_specs["Branches"]-1
67         dct_tree_smaller["Size"] = dct_tree_specs["Size"]/1.61
68         draw_tree(dct_tree_smaller) # RECURSIVE CALL!
```

VIDEO TUTORIAL



[Explanation of user dictionary tree](#)

RELATED CONTENT



[Fractal Generating Software](#)

```
69         # 4 - go back
70         turtle.left(40)
71         turtle.back(dct_tree_specs["Size"])
72     # leaf level reached (level == 0)
73     else: # BASE CASE!
74         # Stamp the leaf
75         turtle.color(dct_tree_specs["Leaves"])
76         turtle.stamp()
77         turtle.color("brown")
78
79
```

The main program code has a couple of changes. We setup the drawing surface as before, but we also add some instructions and change the function call so that it retrieves the values from the user.

```
80 #----- Main Program Starts Here -----#
81
82 # Create a drawing surface
83 drawing_surface = turtle.Screen()
84
85 # Move the turtle
86 turtle.speed(0)
87 turtle.penup()
88 turtle.goto(0,-180)
89 turtle.left(90)
90 turtle.pendown()
91
92 # Setup drawing
93 turtle.color("brown")
94 turtle.width(3)
95 turtle.shape("triangle")
96
97 # Instructions
98 print("Hello this program will draw a tree built to your specifications.")
99
100 # Get User input and draw the tree
101 get_user_tree()
102
103 # exits properly
104 drawing_surface.exitonclick()
```

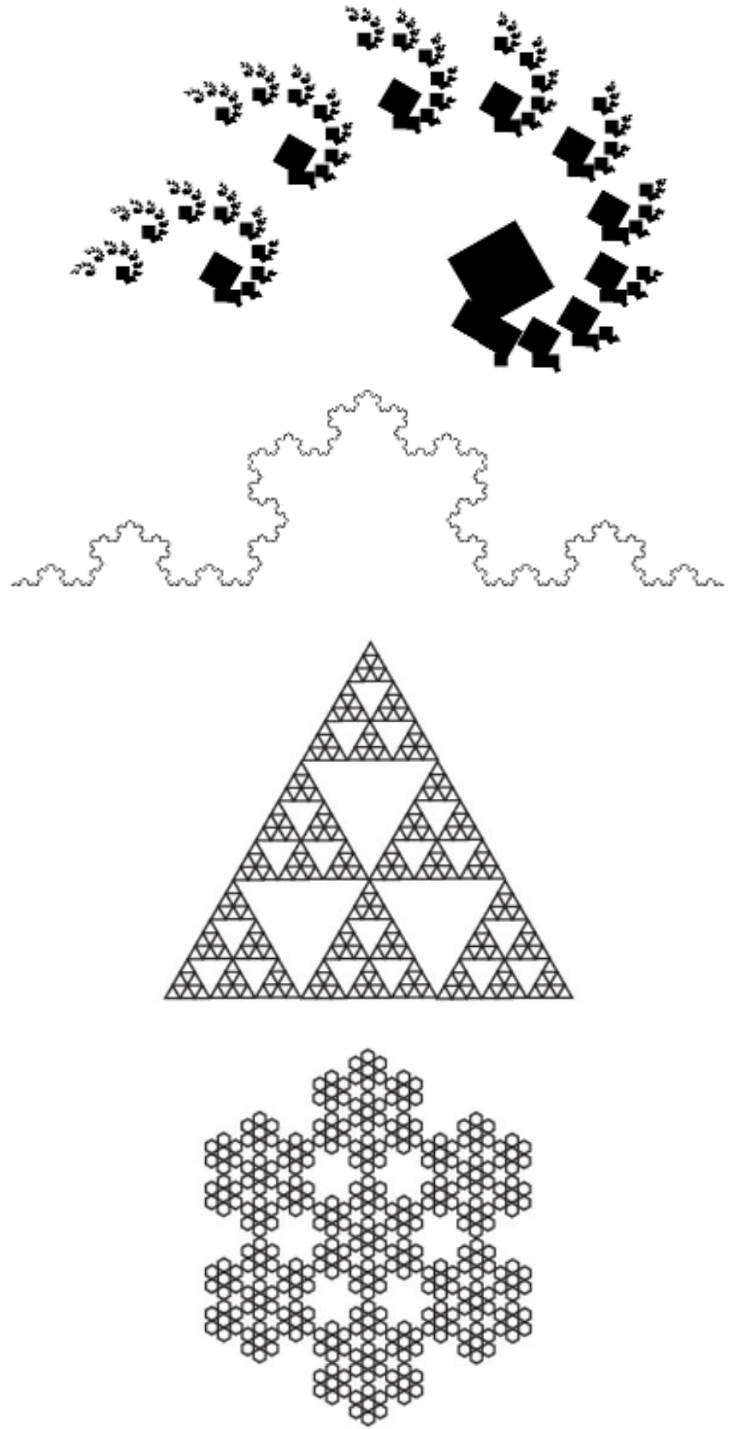
Try it out. Does it work? Watch the video in the left bar for an explanation of the code and to see the application in action.

4.6 Fractals

A fractal is a pattern that reoccurs at progressively smaller scales. The tree we just finished creating with function recursion is an example of a fractal.

Fractals are a naturally occurring phenomenon in nature appearing in leaves, snowflakes, frost, rocks, erosion, and more. In computer programming fractals can be used in the process of resizing images or to reproduce virtual reality environments efficiently. Fractal design can also be applied to data and has even been used in financial tools for analyzing the prices of stocks.

Some visual examples of fractal patterns:



fractal.py – Exercise (5 Marks)

Like the `user_dictionary_tree` assignment, you will create your own drawing that uses at least one recursive function and user input.

The drawing your application makes should be interesting and unique.

Create a file called **fractal.py** in your network folder on (A:) drive.

Your code must include the following:

- Good Style
 - algorithm
 - naming conventions
 - blank line spacing – 2 after each function, 1 between programming blocks
 - programmer commenting
 - functions at the top immediately after import statements
 - program code that isn't inside a function should be written after the functions have been declared
- Information for the user – description of the app and clear instructions for what the app will draw and what the user's input choices are
- At least two functions.
- At least one function will have a dictionary for a parameter.
- At least one dictionary to hold key value pairs for user input.
- A drawing surface and exit statement so that your program does not crash when closing
- Your program must be different from classmates

Note: You can have more than 2 functions and more than one dictionary if you wish.

Evaluation (5 Marks)

/1 Good Style

/1 Minimum of 2 Functions (one recursive) with docstrings

/1 Dictionaries are used properly

/1 Program closes properly using drawing surface and exitonclick

/1 Program is creative and unique and utilizes input from a human to create the fractal

Review Questions:

1.) What would this code output?

```
cafe_prices = {"dark roast": 1.99, "americano": 2.50, "muffin": 2.00}
print(cafe_prices["dark roast"] + cafe_prices["muffin"])
```

2.) What are two components of a recursive function?

3.) When a recursive function calls itself, what must we make sure of?

4.) What is the issue with this code?

```
milkshakes = 10
while milkshakes > 0:
    milkshakes += 1
print(milkshakes)
```

PYTHON CODE CONCEPT

🔗 Functions that return values

4.7 Fruitful Functions

So far, we have learned to create functions with parameters, function calls with arguments, and functions being called from inside another function. The functions we've made have performed tasks but have not resulted in the output of a value that can be used in our program.

It is also possible to have a function return a value as it's purpose. For example, the built-in `.choice()` function in the random module returns a random value from a list of values that it receives.

```
# An example of a fruitful function you've used before

import random

# The random module has defined a function named
# choice that returns a randomly picked element
# from a list argument

# You can store the returned value in a variable to use it
my_choice = random.choice(["TV", "Phone", "Book"])
print(my_choice)

# You can use that returned value directly
print(random.choice(["apples", "oranges", "bananas"]))

# You can use the returned value in a conditional statement
vehicle_list = ["pickup truck", "sedan", "jeep", "motorcycle"]
while random.choice(vehicle_list) != "sedan":
    print(random.choice(vehicle_list))
```

A **fruitful function** is a function that **returns** a value and the function is called from inside an assignment or conditional statement.

```
# An example of a fruitful function you could make yourself
def power(x,y):
    """ Returns a value that is the y power multiple of x """
    return x**y

# Use the value from the fruitful function
print(power(2,3))
```

The function in the example above has a return statement.

When a return statement executes, the function stops running and returns a value. You can have multiple return statements in a function but only one will be executed.

```
# An example of a fruitful function you could make yourself
def power_or_multiple(x,y,operation):
    """ Returns a value that is the y power multiple of x
    or the multiple of x and y depending on the requested operation """

    if operation == "multiply":
        return x * y
    else:
        return x**y

# Use the value from the fruitful function
print(power_or_multiple(2,3,"multiply"))
```

fruitful_demo.py – Practice (2 marks)

Create a **fruitful_demo.py** file and save it to your network (A: drive) folder.

Include the header and algorithm.

```
1  # Draw an exponentially incrementing object using fruitful functions
2  # Author: Your Name
3  # Date: Today's Date
4
5
6  # Get user input for the number of nested squares to create between 8 and 16
7  # for as many squares as the user requests
8  #   get the size of the sides from a power function
9  #   power function gets the current number of the square (i) and a
10 #   float value (1.5) and returns 1.5 ** i
11 #   draw a square with the size value as an argument for the square function
12 #   for i in range(4)
13 #       forward(sidelength)
14 #       right (90)
15
```

Import the turtle module and create this simple example of a fruitful function that returns a float value.

```
16 import turtle
17
18 def power(x,y):
19     """ Returns the value of x ** y """
20     return x**y
21
22
```

Add another function that will draw the square.

```
23 def square(sidelength):
24     """ draws a square the size of the sidelength float value """
25     for i in range(4):
26         mack.forward(sidelength)
27         mack.right(90)
28
29
```

Add the main program code.

```
30 #----- MAIN -----#
31 # Setup the drawing surface and the turtle
32 drawing_surface = turtle.Screen()
33 mack = turtle.Turtle()
34 mack.pencolor("purple")
35
36 # initialize value for number of squares
37 num_of_squares = 0
38
39 # instructions
40 print("Hello! I will draw squares that gets bigger and bigger")
41
42 # Get user input
43 while num_of_squares < 8 or num_of_squares > 16:
44     num_of_squares = int(input("How many squares do you want? (8 to 16) \n>"))
45
46 for i in range(num_of_squares):
47     square(power(1.5,i))
48
49 # close program properly
50 drawing_surface.exitonclick()
```

Test it out and try different values to see what works.

PYTHON CODE CONCEPT

🔗 Sentinel Values

To complete this assignment, modify it like the example on the previous page that has the `power_or_multiple()` function and two return statements.

Note: You might need to change the numbers the user can enter so your drawing still stays on the screen. 😊

4.8 While with Lists and for Input

We have already used `while` for validating input. However here is another example of this concept.

```
# Using While to Validate Input
# Lisa Mulzet
# May 7, 2020

# Create a boolean variable
valid_input = False

# Keep looping if not valid input
while not valid_input:
    # Get input
    answer = input("Do you like cookies (Y/N)? \n>").upper()

    # Update boolean if the input is valid
    if answer == "Y":
        valid_input = True
        print("Cookie monster feels you could be his soulmate.")
    elif answer == "N":
        valid_input = True
        print("Cookie monster can't be your friend.")
```

This method negates the need to initialize the variable that will hold content from the user. The Boolean variable (in this case named `valid_input`) could be utilized several times.

`valid_input` is a Boolean variable that is being used as a **sentinel** or **flag**. A **sentinel value** is a value used to signal the end of a loop.

number_guess.py – Practice (2 marks)

We will now do a little more practice with a while loop to check user input, a while loop with a sentinel, and with using fruitful functions.

Create a **number_guess.py** file and save it to your network (A: drive) folder.

Begin with the header and algorithm.

```
1 # Number Guess Game
2 # Lisa Mulzet
3 # May 8, 2020
4
5 # ALGORITHM
6 # computer_number = call random_number(min,max) function
7 # generate a random number for the user to guess between min 1 and max 100
8 # return random number
9
```



```

10 # give instructions to the user
11
12 # while the game isn't finished...
13
14     # guess = call check_valid(min,max) function
15     # initialize guess_correct to false
16     # while guess_correct is false
17         # ask the user to guess a number between min 1 and max 100
18         # if user_guess > min and user_guess < max
19             # guess_correct = True
20             # return user_guess
21
22 # compare guess with computer_number and count # of guesses
23 # guess_checker(computer_number, user_guess, guess_counter)
24     # guess_counter+=1
25     # if user_guess < computer_number
26         # return ("higher", guess_count)
27     # elif user_guess > number_to_guess
28         # return ("lower", guess_count)
29     # else
30         # return ("Congratulations! You got it.", guess_count)
31

```

We need to import the random module so that we can generate a random number between two values. Then we make a function that has some reusability with parameters for the two values.

```

32 import random
33
34 def random_number(minimum,maximum):
35     """ Generate and return a random number between a
36         minimum and maximum value """
37
38     number = random.randrange(minimum,maximum)
39     return number
40
41

```

We will also create a function that will get the user's guess and only return the user's guess if it is within the minimum and maximum values.

Notice that the while loop is used with **guess_correct** holding a sentinel value that will be used to break out of the iteration when the user provides an acceptable answer.

```

42 def check_valid(minimum,maximum):
43     """ Returns user value only if it is between the minimum and maximum """
44
45     # initialize boolean for while loop input check
46     guess_correct = False
47
48     while guess_correct == False:
49         user_guess=int(input("Please enter a valid number: "))
50         if user_guess >= minimum and user_guess <= maximum:
51             guess_correct = True
52             return user_guess
53
54

```

Once the user's guess has been validated and returned, we can send it to the next function which will compare the user's guess to the computer's random number.

This next function also keeps track of the number of guesses.

```

55 def guess_checker(computer_number, user_guess, guess_counter):
56     """ Compares user_guess with computer_number, advances a guess_counter and
57     returns feedback """
58
59     # advance guess counter
60     guess_counter += 1
61
62     # give feed back for higher/lower/correct and number of guesses
63     if user_guess < computer_number:
64         return ("higher", guess_counter) # this is a list
65     elif user_guess > computer_number:
66         return ("lower", guess_counter)
67     else:
68         return ("Congratulations! You got it.", guess_counter)
69
70

```

It should be noted that functions can only return one thing at a time. In the `guess_checker` function it looks like it is returning more than one thing. However, it is returning a list which is a single object (like a box of chocolates) that can contain multiple elements (the chocolates).

Next we will setup variables that have module level scope. The Boolean variable, `finished`, is the sentinel for our main program loop.

```

71 # ----- Main Code ----- #
72
73 # generate a random number between 1 and 100
74 computer_number = random_number(1,100)
75
76 # initialize a guess counter
77 guess_counter = 0
78
79 # initialize a sentinel to stop the program
80 finished = False
81

```

Give instructions to the user.

```

82 # print instructions for the user
83 print("\n Welcome to the number guessing game.")
84 print("I'm thinking of a number between 1 and 100. What is it? \n")
85

```

Our program will keep running until the user guess matches the random number that the computer has generated.

```

86 while not finished:
87
88     # get valid user input
89     guess = check_valid(1,100)
90
91     # get feedback and number of guesses so far
92     response = guess_checker(computer_number, guess, guess_counter)
93
94     # update guess counter
95     guess_counter = response[1]
96
97     if response[0] == "Congratulations! You got it.":
98         print("")
99         print(response[0] + " You made " + str(response[1]) + " guesses")
100         finished = True
101     else:
102         print("")
103         print("Uh oh, you need to guess " + response[0] + ".")
104         print("So far you've made " + str(response[1]) + " guesses.")
105         print("")

```

`response` is a list that is receiving the returned list from the `guess_checker` function.

`response[0]` is a string element in the list and it could be equal to “lower” or “higher” or “Congratulations! You got it.”

`response[1]` is an integer element in the list and holds a running total for the number of guesses.


Run your code and make sure it is working properly.

Now modify it so that if the user enters 0, finished will be set to True.


Hint: You will need to change code inside the `check_valid` and `guess_checker` functions. You will also need to update the user instructions and your main program code to handle a fourth `response[0]` value.

When you’ve completed the program, and the user is now allowed to give up by typing 0, it’s ready to be marked. 😊

PYTHON CODE CONCEPT

 [Concatenation and Repetition](#)

VIDEO TUTORIAL

 [Concatenate Must be Done with Like Types](#)

Another concept that we have looked at recently is concatenation. **Concatenation** allows us to add more content to lists or strings using `+` in Python (some programming languages use `&` for concatenation).

A while loop combined with concatenation is a way to build a list with user input.

It’s important to note that concatenate only works with like types. Lists can only be concatenated with lists. Strings can only be concatenated with strings. Watch the video explanation to the left.

flower_field.py – Practice (2 marks)

We will now practice a little concatenation, using `while` and lists, to create a field of flowers.

Create a **flower_field.py** file in your network save location (A: drive).

Type the header and algorithm.

```

1 # Field of Flowers
2 # Name: Your Name
3 # Date: May 11, 2020
4
5 # Setup the field
6 # Create color lists for the flower centers and petals
7 # Create a list of flowers until the user wishes to stop
8 #   Tell user what color choices they have
9 #   Tell the user what number of flower they are on
10 #   Randomly generate the color for the flower center
11 #   Ask the user to pick a petal color from an approved color list
12 # Send the two lists (center_color) and (petal_color) to draw_flowers function
13 #   for each petal color in the petal_color list
14 #   pick a random location from a random_location function
15 #   Draw each flower
16

```

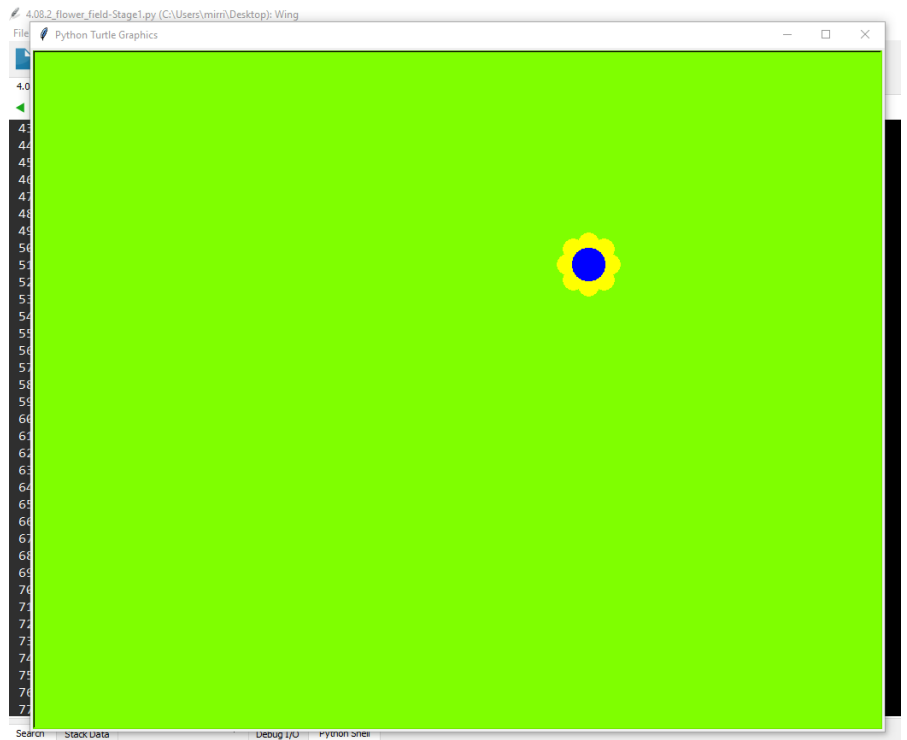
Let's do this incrementally and make sure our graphics will load properly and our flower will draw on the screen.

```

17 import turtle
18
19
20 def draw_flowers(petal_colour, center_colour, location):
21     """ Receives a color list for the flower petals and a color list for
22         the flower centers. For each color in the flower petal list, a flower
23         will be drawn at a random location in the window. """
24
25     bee = turtle.Turtle() # our flower artist is a bee :)
26     bee.ht() # hide bee
27     bee.speed(10)
28
29     # move to new random location
30     bee.penup()
31     bee.setx(location[0])
32     bee.sety(location[1])
33
34     # get the petal color in the list at index i
35     bee.color(petal_colour)
36
37     # draw the petals
38     for side in range(8):
39         bee.forward(25)
40         bee.dot(25)
41         bee.backward(25)
42         bee.right(45)
43
44     # get the center color in the list at index i and draw the center
45     bee.color(center_colour)
46     bee.dot(40)
47
48 # ----- MAIN CODE ----- #
49
50 # Constants local to module
51 WIDTH = 1000
52 HEIGHT = 800
53
54 # setup the screen
55 field = turtle.Screen()
56 field.setup(WIDTH,HEIGHT,20,20)
57 field.bgcolor("#7FFF00")
58
59 # draw the flowers with user input
60 draw_flowers("yellow","blue",(150,150))
61
62 # exit the program properly
63 field.exitonclick()
64

```

If it's working properly it should look like this.



For the next stage let's have our flower appear at a random location by adding a function that returns a random x and y coordinate as a list.

Begin by adding an import statement for the random module.

```
17 import turtle
18 import random
```

Add a function to generate a **random_location**. It will have two parameters, so it knows how big the screen is. These values were listed as constants **WIDTH** and **HEIGHT** in the main portion of the program and we will use them when this function is called.

```
19
20 def random_location(window_width, window_height):
21     """ Receives the window width and height. Randomly generates and returns
22         a coordinate list for x and y """
23
24     # center of turtle screen is 0,0
25     # generate x coordinate
26     x_coord_min = (-1)*((window_width-40)/2) # -40 buffer to stay on screen
27     x_coord_max = (window_width/2)
28     x = random.randrange(x_coord_min, x_coord_max)
29
30     # generate y coordinate
31     y_coord_min = (-1)*(window_height/2)
32     y_coord_max = ((window_height-40)/2)
33     y = random.randrange(y_coord_min, y_coord_max)
34
35     # return coordinates as a 2 item list
36     return(x,y)
37
38
```

To test this function, we will need to assign the list value for the random flower location inside our `draw_flowers` function.

Inside the `draw_flowers` function after `bee.penup()` assign the value coming from the fruitful function, `random_location`, to the list called `location`.

```
48     # move to new random location
49     bee.penup()
50     location = random_location(WIDTH,HEIGHT) # random location
51     bee.setx(location[0])
52     bee.sety(location[1])
```

Change the parameters for `draw_flowers` because the location values are now being generated inside the function.

```
39 def draw_flowers(petal_colour, center_colour):
40     """ Receives a color list for the flower petals and a color list for
41         the flower centers. For each color in the flower petal list, a flower
42         will be drawn at a random location in the window. """
```

Change the function call for `draw_flowers` so that you are only sending arguments for color.

```
80 # draw the flowers with user input
81 draw_flowers("yellow","blue")
```

Test it out! If it has worked correctly, each time you run the program a single flower will be drawn at a new location each time.

Now let's add a function that creates color lists for the flower petals and centers.

We will have to modify our `draw_flowers` function so that it is working with the `petal_colour` and `center_colour` as though they were lists. It will loop through each petal in the `petal_colour` list and generate a flower each time.

We will need to add a for loop and use the iteration count `[i]` for list index values.

```

39 def draw_flowers(petal_colour, center_colour):
40     """ Receives a color list for the flower petals and a color list for
41         the flower centers. For each color in the flower petal list, a flower
42         will be drawn at a random location in the window. """
43
44     # for each color in the petal_colour list draw a flower
45     for i in range(0, (len(petal_colour))):
46         bee = turtle.Turtle() # our flower artist is a bee :)
47         bee.ht() # hide bee
48         bee.speed(10)
49
50         # move to new random location
51         bee.penup()
52         location = random_location(WIDTH, HEIGHT) # random location
53         bee.setx(location[0])
54         bee.sety(location[1])
55
56         # get the petal color in the list at index i
57         bee.color(petal_colour[i])
58
59         # draw the petals
60         for side in range(8):
61             bee.forward(25)
62             bee.dot(25)
63             bee.backward(25)
64             bee.right(45)
65
66         # get the center color in the list at index i and draw the center
67         bee.color(center_colour[i])
68         bee.dot(40)
69

```

Then you will add a `get_colours()` function, after the `draw_flowers()` function, that asks the user for input and concatenates the lists for `center_colour` and `petal_colour`.


There are two mystery lines you must complete for this function to work properly. Can you do it?

```

71 def get_colours():
72     """ Generates a color list for flower centers by randomly choosing a
73         from a list of colour choices. Builds a list of user input for the
74         flower petal colours. Sends both lists to a function that draws
75         multiple flowers. """
76
77     # initialize
78     done = False
79     petal_colour = []
80     center_colour = []
81     flower_count = 0
82
83     # allowed colors for centers and petals
84     colour_choices = ["red", "orange", "yellow", "green", "blue", "indigo",
85                       "violet", "black", "white", "purple", "crimson", "olive"]
86
87
88     # instructions
89     print("Our invisible bee will make as many flowers as you like. You get")
90     print("to choose the petal colors. When you want to stop making")
91     print("flowers just press enter instead of entering a color")
92
93     # loop until sentinel triggers a break
94     while not done:
95
96         # keep count of flowers so user knows how many
97         flower_count+=1
98
99         # pick random center and add it to a list for flower center
100        middle = colour_choices[random.randrange(len(colour_choices))]
101        # MYSTERY LINE OF CODE. WHAT GOES HERE? # list concatenation
102

```

VIDEO EXAMPLE

 [flower field completed](#)

```
102
103     # give instructions
104     print("Flower #" + str(flower_count))
105     print("The center color will be: " + middle)
106     print("Choose a petal color from this list:")
107     print(colour_choices)
108
109     # get petal color
110     colour = input("Petal Colour: ")
111     if colour in colour_choices: # allowable petal color
112         # MYSTERY LINE OF CODE. WHAT GOES HERE? # list concatenation
113         elif colour == "": # sentinel has been entered
114             done = True
115         else: # petal color not recognized
116             print("Invalid color.")
117             flower_count -= 1
118             # now center_colour will have more items than petal_colour list
119             # so we should remove the last item from the list using .pop()
120             center_colour.pop()
121
122     # send the petal and center color lists to the flower drawing function
123     draw_flowers(petal_colour, center_colour)
124
125
126 # ----- MAIN CODE ----- #
```

Change the function call in the main code so that it is calling `get_colours()` instead of `draw_flowers()`.

```
124 # ----- MAIN CODE ----- #
125
126 # Constants local to module
127 WIDTH = 1000
128 HEIGHT = 800
129
130 # setup the screen
131 field = turtle.Screen()
132 field.setup(WIDTH,HEIGHT,20,20)
133 field.bgcolor("#7FFF00")
134
135 # draw the flowers with user input
136 get_colours()
137
138 # exit the program properly
139 field.exitonclick()
```

If you did it correctly it should work like the demo video to the left.

Review Questions:

1.) How many circles does this draw?

```
import turtle

bob = turtle.Turtle()

def draw_circle(radius):
    if radius > 20:
        bob.circle(radius)
        draw_circle(radius/2)

draw_circle(120)
```


2.) What is the output of this code if the input is **Peter**?

```
my_name = input("What's your name? ")
secret = ""
for i in range(len(my_name)):
    token = my_name[len(my_name) - 1 - i]
    if token.lower() in ["a", "e", "i", "o", "u"]:
        token = token.upper()
    secret += token
print(secret)
```

3.) What would this code output if the user tried to enter the following?

a<enter>be<enter>sea<enter>dea<enter>ee<enter>eff<enter>

```
exclamation = ""
while len(exclamation) < 10:
    exclamation += input("Enter some character:")
print(exclamation)
```

show_off.py – Exercise (5 Marks)

In this unit we've learned about the following code concepts:

- functions with parameters
- recursive functions
- fruitful functions
- dictionaries
- while with a sentinel
- while to check input
- list concatenation

Create a file called **show_off.py** in your network folder on (A:) drive.

Build an application that uses 4 of the coding concepts in the list above. It can use turtle graphics or it can be text based with no graphics.

In your programmer comments be sure to note which concept you are demonstrating so that your teacher can quickly find

Evaluation (5 Marks)

/1 Good Style & Algorithm

/1 Mark for each of the concepts listed above that you used

Unit Coding Vocabulary and Concept Links:

Functions

A function is a block of code which only runs when it is called. Functions are used to reduce repetitive code and perform a specific task multiple times.

Links: [Introduction to Functions](#), [Functions in more Detail](#)

Parameters

Parameters are information that a function receives, and they are named in the parenthesis of the function header.

Arguments

When a function is called that has parameters, arguments are the information that is sent inside the parenthesis of the function call. Arguments must be in the same order, and of the same data type, as the parameters that are in the function that is being called.

Recursion (Recursive Functions)

Recursive functions are a form of iteration where the function calls itself. To avoid an infinite loop, a recursive function must have a base case.

Base Case

In a recursive function, the base case is a condition that the repeating function gets closer and closer to. When the base case is reached, the recursive function call will stop repeating.

Fruitful Functions

Dictionaries

While Loop

While Loop with a sentinel

While Loop to check input

Concatenation with Lists

Credit:

The curriculum concept and units in this course were created by Angelica Lim who is an Associate Professor at SFU ([Linked In Profile](#)). She is involved in SFU's AI and robotics program in the Rosie Lab (<https://www.rosielab.ca/>).

Many of the coding examples and exercises in this course are Angelica's creations.

The course curriculum has been adapted for students at Centennial Secondary school by Lisa Mulzet who has created this PDF textbook for use by her students.

