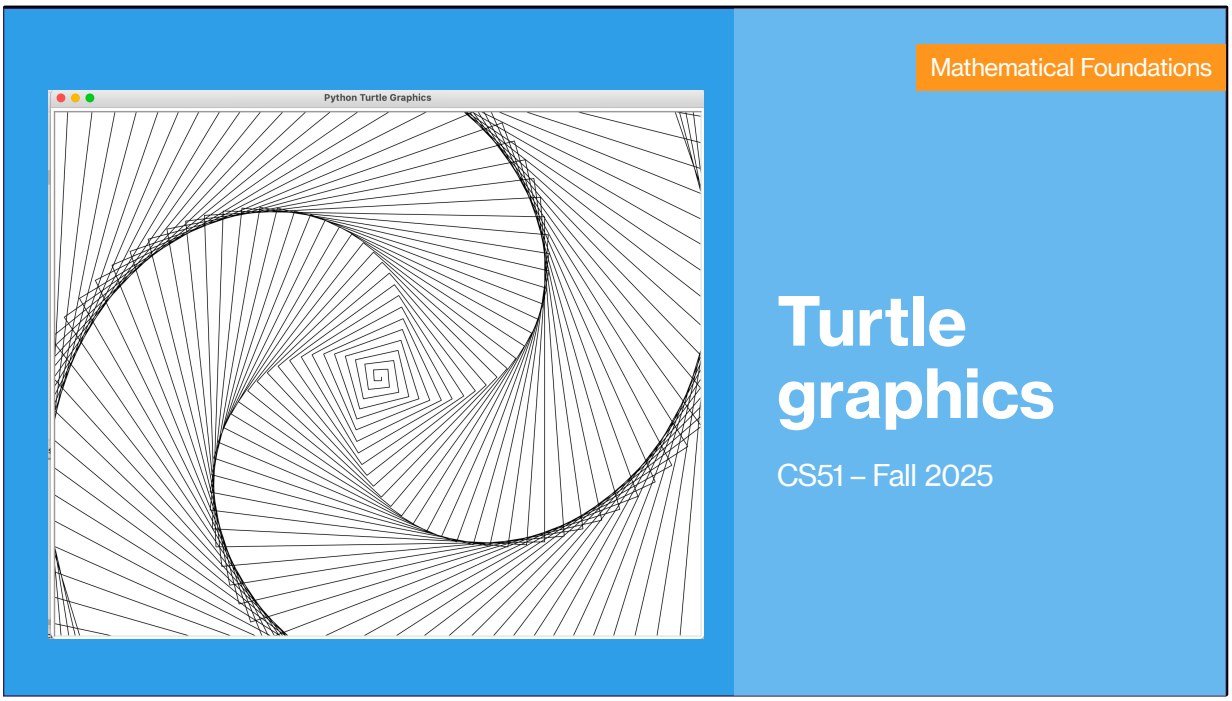


Today, we will conclude the unit on mathematical foundations of computer science and we will explore a more creative side of Python.

Python modules

- **Module**: a collection of functions and variables allowing us to share code with others.
- For example, there is a module called `math` that has many math functions you might want.
  - We can look at the documentation for this module by going to https://docs.python.org/3/ and searching `math` which will lead you to https://docs.python.org/3/library/math.html#module-math
- Some examples of useful functions and variables in the `math` module:
  - Variations of logs with different bases (e.g., `log`, `log2`, `log10`)
  - `sqrt` for calculating the square root
  - trigonometric functions (e.g., `cos`, `sin`, `tan`)
  - Constants (e.g., `pi`, e, etc.)

2

In Python, a module is a collection of functions and variables. Modules allows us to use code that other people have written and enable us to share our own code. For example, there is a module called math that has many of the math functions you might want. If you look up the official documentation of Python, you will discover that the math module offers you access to all sorts of functions, such as logarithmic ones, square root, trigonometric functions, and constants such as PI, e, etc.

Importing modules

- If we want to use a module, we need to tell the program to "import" it in our program.
- Code to import all functions and variables from `math` module:
  - `from math import *`
- This statement has multiple components:
  - `from` is a keyword,
  - `math` is the name of the module,
  - `import` loads the module into our program,
  - `*` means everything, i.e. all functions and variables included in the `math` module.
- Just replace `math` with the name of the module you want to import.

3

If we want to use a module, we need to tell the program to include it with our program. We would type from math import *.
This statement has multiple components:
from is a keyword,
math is the name of the module,
import loads the module into our program,

- means everything, i.e. load everything included in the math module.
- Just replace `math` with the name of the module you want to import.

`random` module

- The `random` module generates pseudo-random numbers.
- If you want truly random numbers, check out http://www.random.org/

4

Another module that will be handy is the the random module. The random module contains functions that generate pseudo-random numbers. Why pseudo (==not genuine)? The numbers are not technically random, they are generated by an algorithm but for most purposes, including ours, they are random enough. If you want truly random numbers check out the website http://www.random.org/

Useful `random` functions

- `random()` - returns a random float between 0 and 1.
- `uniform(a, b)` - returns a random float between a and b.
- `randint(a, b)` - returns a random integer between a and b.

5

Some functions that you might find useful are random (returns a number between [0,1), uniform (returns a float between [a,b] or [a,b) depending on rounding), and randint (returns an int between [a,b]), as well as samples from well-known distributions.

Importing only one function

- Let's say we only want to use the `randint` function.
- Rather than importing everything (*) we will be specific:
  - `from random import randint`

- What would the following piece of code do?

```
for i in range(100):
    print(randint(0,10))
```

- Prints 100 random integers between 0 and 10 (inclusive)

6

Let's say we only want to use the `randint` function.
. Instead of loading all functions, we can be specific: from random import randint.

`turtle` module

- The `turtle` module controls the movements of a "turtle" (really, an arrow), such as:
  - `forward(distance)`: Move the turtle forward by the specified distance.
  - `backward(distance)`: Move the turtle backward by distance, opposite to the direction the turtle is headed without changing the turtle's heading.
  - `right(angle)`: Turn the turtle right by angle units.
  - `left(angle)`: Turn turtle left by angle units.
  - ...and many others. Check the documentation and more practice in the upcoming assignment.
- As the turtle moves, it draws a line, so by controlling it we can draw on the screen!

7

Time to have fun with a cool module called turtle...The turtle module implements a set of commands similar to the Logo programming language
The basic idea is that you control the movements of a turtle (in our case, it will be an arrow) through basic commands such as:
forward
backward
right
left
...and many others
As the turtle moves, it draws a line behind it, so by giving it different commands, we can draw things on the screen!
Check the documentation for the turtle class online
You'll be getting more comfortable with this documentation in the upcoming assignment.

Open VS Code

- Go to the Python shell at the bottom
- python3
- >>> from turtle import *

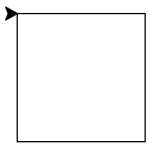
8

OK let's give it a try together. If you have your laptop, open VS Code and type python3.
Then from turtle import *

Note that if you try to import the turtle module on macOS and receive the following error "import _tkinter # If this fails your Python may not be configured for Tk", you will need to go https://www.python.org/downloads/release/python-3140/ download again python, install it, and then open a Terminal and write
PATH="/Library/Frameworks/Python.framework/Versions/3.14/bin:${PATH}"
export PATH

Let’s move our turtle

- How would you create a square by 20 units?
  - `forward(20)`
  - `right(90)`
  - `forward(20)`
  - `right(90)`
  - `forward(20)`
  - `right(90)`
  - `forward(20)`



9

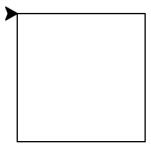
Note that if you try to import the turtle module on macOS and receive the following error "import _tkinter # If this fails your Python may not be configured for Tk", you will need to go https://www.python.org/downloads/release/python-3140/ download again python, install it, and then open a Terminal and write PATH="/Library/Frameworks/Python.framework/Versions/3.14/bin:${PATH}" export PATH

Let’s say we want to create a square where each side is 20 units. How we would do it? We just want to repeat four times forward(20) and right(90)

Let’s move our turtle

- Packaging our square drawing code into a function:

```
def square(length):
    forward(length)
    right(90)
    forward(length)
    right(90)
    forward(length)
    right(90)
    forward(length)
square(100)
```



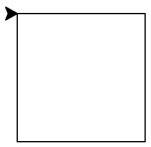
10

Let’s put our code in a function to make it more reusable. If we run this code, we can call square, e.g., as square(100) and see the following turtle trail. This seems like a lot of repetitive typing. Let's say we can tell the turtle to repeat some statements, would there be a better way of saying in English that we want to create a square? go forward some length and then turn right, repeat this 4 times

Let's move our turtle

- A less verbose function to draw a square:

```
def iterative_square(length):
    for i in range(4):
        forward(length)
        right(90)
iterative_square(20)
```



11

We can of course make our code less verbose by putting it in a for loop

mystery1 function

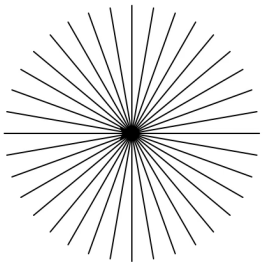
```
def mystery1():
    for i in range(36):
        forward(100)
        backward(100)
        right(10)
```

12

What do you think this mystery_function does? Think about it and if you are not sure type it in your shell.

`simple_star` function

```
def simple_star():
    for i in range(36):
        forward(100)
        backward(100)
        right(10)
```



13

It draws a 36-sided star (or asterisk)

mystery2 function

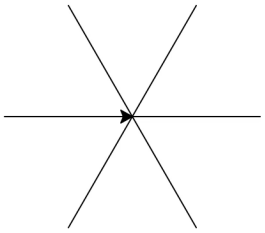
```
def mystery_2(par1, par2):
    var = 360 / par2
    for i in range(par2):
        forward(par1)
        backward(par1)
        right(var)
```

14

What about this mystery function?

A star with a different number of spokes

```
def asterisk_star(length, spokes):
    angle = 360 / spokes
    for i in range(spokes):
        forward(length)
        backward(length)
        right(angle)
```



15

It creates a star/asterisk with a different number of spokes. We first figure out how we have to space the spokes. We do a for loop over the number of spokes. At each iteration we draw a spoke, go backwards for the next spoke, rotate right based on the angle we calculated. Here's the result of asterisk_star(100, 6)

mystery3 function

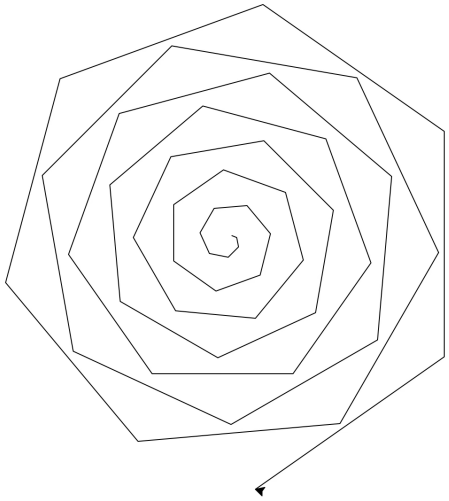
```
def mystery3():
    for i in range(50):
        forward(i * 5)
        right(55)
```

16

What about this mystery function?

`simple_spiral` function

```
def simple_spiral():
    for i in range(50):
        forward(i * 5)
        right(55)
```



It draws a simple spiral! Look at the code: The variable i will keep count of how many times we need to get in the loop. For small loops (i.e. just a handful of statements), it's common to just name the counter variable as i which stands for index. Each time we get in the loop, the length of the edge drawn will be longer by 5: 0, 5, 10, 15, 20, … and it will be at a 50 degree angle. If it is less than 90 degrees, it will spiral out, and above it, it will spiral in.

mystery4 function

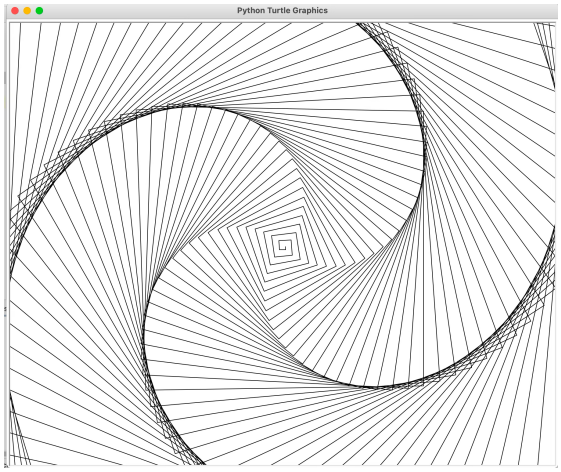
```
def mystery4(par1, par2):
    for i in range(par1):
        forward(i * 5)
        right(par2)
```

18

What about this mystery function

`spiral` function

```
def spiral(sides, angle):
    for i in range(sides):
        forward(i * 5)
        right(angle)
```



19

The spiral function is similar to simple_spiral, however now we've parameterized the length of the sides and the angle. A good example, is side = 200 and angle = 89

mystery5 function

```
def mystery5(par1, par2):
    var = 360 / par2

    for i in range(par2):
        circle(par1)
        right(var)
```

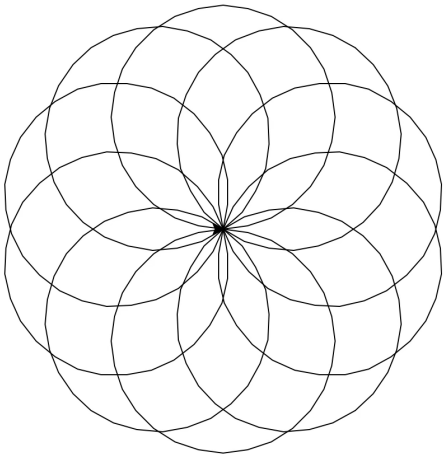
20

What about this function?

rotating_circles function

```
def rotating_circles(radius, num):
    angle = 360 / num

    for i in range(num):
        circle(radius)
        right(angle)
```



21

Finally, rotating_circles is another fun function that draws “num” circles, each one rotated "angle" degrees from the previous one. This is the result of the rotating_circles(100, 10) call.

mystery6 function

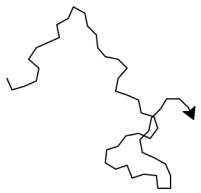
```
def mystery6 (par1, par2):
    for i in range(par1):
        var = randint(-90, 90)
        right(var)
        forward(par2)
```

22

What about this function?

`walk` function

```
def walk(num_steps, step_size):
    for i in range(num_steps):
        angle = randint(-90, 90)
        right(angle)
        forward(step_size)
```



23

It draws a random walk: It turns a random angle between -90 and 90 and steps forward some step size for num_steps.

mystery7 function

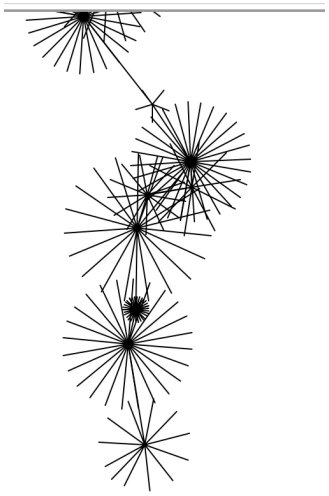
```
def mystery7():
    for i in range(10):
        var1 = randint(5, 30)
        var2 = randint(10, 60)
        var3 = randint(-90, 90)
        var4 = randint(20, 100)
        right(var3)
        forward(var4)
        asterisk_star(var2, var1)
```

24

What about this mystery function?

pretty_picture function

```
def pretty_picture():
    for i in range(10):
        spokes = randint(5, 30)
        length = randint(10, 60)
        angle = randint(-90, 90)
        move = randint(20, 100)
        right(angle)
        forward(move)
        asterisk_star(length, spokes)
```



25

It draws a line of length between 10 and 60 at an angle between -90 and 90 then draws a star of length between 10 and 60 and with “spokes” spokes. It does that 10 times. Pretty, right?

setcolor_random function

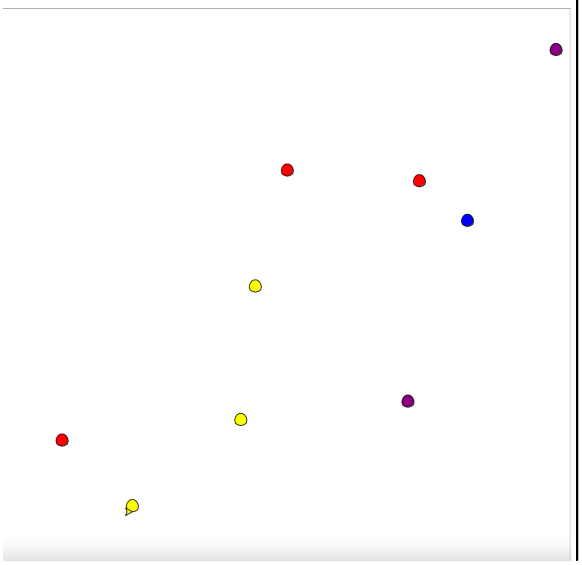
```
def setcolor_random():
    color = randint(1, 4)
    if color == 1:
        fillcolor("blue")
    elif color == 2:
        fillcolor("purple")
    elif color == 3:
        fillcolor("red")
    else:  # color == 4
        fillcolor("yellow")
```

26

What about the setcolor_random function? It randomly picks between blue, purple, red and yellow (instead of based on x, y). Any ideas on how could we get this behavior? We will use random.randint to select a number between 1 and 4. We will save this number and use it in an if-elif-else statement. We MUST save this number to a variable and not try and do your if/else statement based on new calls to random.randint!!!

`add_circles` function

```
def add_circles(number):
    x_range = int(window_width() / 2)
    y_range = int(window_height() / 2)
    for i in range(number):
        x = randint(-x_range, x_range)
        y = randint(-y_range, y_range)
        setcolor_random()
        pu()
        goto(x, y)
        pd()
        begin_fill()
        circle(8)
        end_fill()
```



It picks random x and y coordinates to draw a circle. It uses the randint function which we have seen before. Now how are the colors chosen? Each quadrant of the xy-axes is a different color. Wait, how can we do this? We will have to ask a question about x and y.

Revisiting creating a spiral

```
def spiral(length, levels):
    if levels == 0:
        dot()
    else:
        forward(length)
        left(30)
        spiral(0.95 * length, levels – 1)
```

28

Let’s revisit the spiral function written now in a recursive manner

Revisiting `spiral` function

- Let's assume I called spiral(80, 50)
- When does it stop?
  - When levels = 0.
    - We put a dot there to make it explicit.
- Repeat 50 times:
  - forward length
  - left 30
  - reduce length by 5%

```
def spiral(length, levels):
    if levels == 0:
        dot()
    else:
        forward(length)
        left(30)
        spiral(0.95 * length, levels - 1)
```

29

Revisiting `spiral` function

- What if we wanted to end up back at the starting point, but not pick the pen up?
- We could trace our steps backwards.

```
def spiral(length, levels):
    if levels == 0:
        dot()
    else:
        forward(length)
        left(30)
        spiral(0.95 * length, levels – 1)
        right(30)
        backward(length)
```

30

What if we wanted to end up back at the starting point, but not pick the pen up?
We could trace our steps backwards.

Assume that the recursive call returns back to its starting point. What would we need to do to make sure that our call returned back to the starting point?

Add the following after the recursive call:

```
right(30)
backward(length)
```

if we run it now, we draw the spiral all the way down, and then we retrace backwards.:

each call to `spiral` retraces its own part after the recursive call.
the call stack keeps track of each of the recursive calls.

`broccoli_demo` function

1. Define what the header function is:
   - `broccoli(x, y, length, angle)`
2. Define the recursive case:
   - broccoli is a line with three other broccolis at the end:
     - one directly straight out
     - one 20 degrees to the left
     - one 20 degrees to the right
   - the three other broccolis should be smaller/shorter than the current

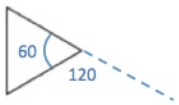
`broccoli_demo` function

3. Define the base case:
   - in each case, the length of the broccoli to be drawn gets shorter.
   - We stop at `length < 10` and place a yellow dot
4. put it all together!
   - Check the base case first:
     - if `length < 10`
       - Draw a yellow dot.
   - Otherwise:
     - draw three smaller broccolis at different angles.

- `new_x` and `new_y` are the ending coordinates of the line being drawn. We save them because after the first recursive call to broccoli the turtle won't be in the same place.

Practice time

- Write a function called `triangle` that draws an equilateral triangle (i.e., a triangle with all three sides the same length).
- Your function should take three parameters: the x and y coordinates where it should be drawn, and the length of the side of the triangle.
- The triangle should be drawn so that the left edge of the triangle is vertical.



33

Write a function called `triangle` that draws an equilateral triangle (i.e., a triangle with all three sides the same length). Your function should take three parameters: the x and y coordinates where it should be drawn, and the length of the side of the triangle. The triangle should be drawn so that the left edge of the triangle is vertical.

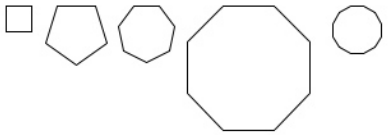
For those rusty on geometry, the interior angles of an equilateral triangle are all 60 degrees, which means the angle between a straight line drawn from one side and the adjacent side is 120 degrees.

Practice time

```
def triangle(x, y, size):
    pu()
    goto(x,y)
    setheading(-30)
    pd()
    forward(size)
    right(120)
    forward(size)
    right(120)
    forward(size)
```

Practice time

- Write a function called `polygon` that draws a polygon.
- An n-sided polygon has n equal-length edges, and the angle between a straight line drawn from one side and the adjacent side is 360/n.
- Your function should take 4 parameters: the x and y location of the polygon, the number of sides, and the length of each side.
- The top edge of your polygon should always be perfectly horizontal.
- Again, the x and y coordinates may indicate any point on the polygon, so pick whatever seems easiest. But you can't just hard-code the number of sides. Instead, you'll have to use a loop.



35

Write a function called `polygon` that draws a polygon.
An n-sided polygon has n equal-length edges, and the angle between a straight line drawn from one side and the adjacent side is 360/n.
Your function should take 4 parameters: the x and y location of the polygon, the number of sides, and the length of each side.
The top edge of your polygon should always be perfectly horizontal.
Unlike the triangle, where you know when you're writing the function exactly how many sides the object will have, for the polygon, you can't just hard-code the different line segments. Instead, you'll have to use a loop. .
For example, below are some polygons of differing numbers of sides and sizes. All can be drawn with the polygon function.

Answer

```
def polygon(x, y, sides, size):
    pu()
    goto(x,y)
    setheading(0)
    angle = 360.0/sides
    pd()
    for i in range(sides):
        forward(size)
        right(angle)
```

Code

turtle-examples.py

turtle-recursion.py