Quick Intro to Turtle Graphics with Python

Author: Joe Seiler

01/26/2018

Objectives:

- Turtle Graphics Module
- Recognizing Patterns
- Python Loops
- Reusing Code
    - Encapsulation
    - Generalization

What is Turtle Graphics?

Python's turtle program (or **module**), is a program that holds a slew of functions waiting to be used. All we have to do is put `import turtle` towards the top of any python file we create and now, we can write instructions that display images using turtle graphics! Rather than explain the turtle module from scratch, take a look at the snippet below from the official python documentation on Turtle Graphics:

"Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an import turtle, give it the command turtle.forward(15), and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command turtle.right(25), and it rotates in-place 25 degrees clockwise." -Python 3.3.7 Doc

Getting Started

If you do not have access to Python at home, you can download it free from python.org, or in an internet browser via PythonAnywhere.com. For either options, you can follow instructions for helping you set up your environment here.

Create a new Python file, filename.py. You can copy and paste the first bit of code below to get started. From here, you can follow along by adding to your program from the steps and descriptions that follows, or create new Python files for each section.

Here's the first program we wrote together using turtle graphics:

```
import turtle

# name your turtle
frank = turtle.Turtle()

# tell your turtle where to go
frank.fd(100)
frank.lt(90)
frank.fd(100)
```

```
# so the window doesn't disappear right away, add:
turtle.mainloop()
# or turtle.exitonclick()
```

Think of the "turtle" as a pen, marker, etc. We are giving this writing tool instructions to draw lines on a separate window (or piece of paper) --similar to our brain giving instructions to our hand to write or draw

Above, we called **methods** to give our turtle instructions on where to go. Methods are like functions that live inside an **object**, but use different syntax.

*More on **objects** later. For now, just think of methods like a function i.e. it has a name defined, a set of operations it can perform, and we can call its name to have it perform those operations.*

Methods are conjured up using dot-notation (a period). Calling a method is like making a request (*or command*). You are telling frank (or whatever you named your turtle) to move forward, x-amount and turn, x-amount, and so on.

Challenge: Add instructions to the right angle you just drew to display a complete square.

You may have ended up with a list of commands like this:

```
frank.fd(100)
frank.lt(90)
frank.fd(100)
frank.lt(90)
frank.fd(100)
frank.lt(90)
frank.fd(100)
frank.lt(90)
```

Recognizing Patterns

Do you see a pattern anywhere in the list of commands above? You should!

Being able to recognize patterns is a useful skill in programming (in general too).

We have the `forward` and `turn` command repeat, four times. We can make this program a little more efficient. Anytime we recognize code that repeats, we can look at using a loop to shorten our code. Our goal is to draw a square... We know there are four sides (and corners) to a square. We already see too, we have two lines of repeating code, the forward command and the turn command.

Don't **R**epeat **Y**ourself, keep **DRY**!

Python `for` loop

One way to repeat code in Python is to use a `for` loop. We can specify how many times we want a block of code to repeat. Substitute your instructions to make a square with the loop below:

```
```

```
for i in range(4):
    # body of code for the "for" loop starts here --notice the indentation
    frank.fd(100)
    frank.lt(90)
```

a `for` statement is a loop because the flow of execution runs through the *body* (or, whatever is indented below), then loops back to the top. In this case, it does so 4 times.

Python `while` loop

We can also use another *repeat* technique, Python's `while` loop. This will run "as long as", or *while* a certain condition is true. As you tweak some of this, you can see how you can create some cool output with ease.

```
while True:
    t.fd(100)
    t.lt(95)
```

Make a Reusable function

We can make our code even more efficient by creating our function. We will make our code more *reusable* by doing this. A function in Python has three parts: a *name*, *parameters*, and a *body*.

```
import turtle

frank = turtle.Turtle()

def square(t):                                  # 'square' is the name of the
function, and 't' is the parameter
    """Reusable function to draw a square."""       # Here, starts the function
body
    for i in range(0, 4):
        t.fd(100)
        t.lt(90)

square(frank)
turtle.mainloop()
```

It might seem silly to do this at first, but what if we wanted to have more than one turtle object draw out the same thing? If we didn't define a function, we would find ourselves writing more code than we need to. You can check this out by defining a couple more turtle variables and calling the function with their names:

```
import turtle

# define turtle variables here
frank = turtle.Turtle()
```

```
marie = turtle.Turtle()
albert = turtle.Turtle()

def square(t):
    """Reusable function to draw a square."""
    for i in range(0, 4):
        t.fd(100)
        t.lt(90)

# call the function and pass the name of each turtle as an argument
square(frank)
square(marie)
square(albert)

turtle.mainloop()
```

By wrapping up a piece of code and putting it inside a resuable function we are introducing **encapsulation**. Why? -to stay **DRY** (Don't Repeat Yourself), or reuse code to write less code, `"t"` can be any turtle. We call the same function; except, we pass the new turtle's name as an argument inside the parentheses of the new function. When we get to having more than one turtle use the function we created, we don't need to concern ourselves at this point with the code that makes the function do what it does; we're just happy it works. We only need to concern ourselves with creating multiple turtles which use the same function, and pass the arguments we want. We are communicating just the information that needs to be exchanged. In other words, we encapsulate the details inside the function. This is a useful tool to keep in mind as we get into larger programs. This idea of hiding the unnecessary details to focus on the goal at hand is known as **abstraction**.

We also introduce ourselves with some more important tools found in languages besides just Python: **parameters** and **arguments**.

In the last exercise we also **generalized**, we made our code usable across more situations by adding the `t` parameter. We can further *generalize* by adding a another parameter to the mix, *length*. By adding this, we can have the output size change each time if we wanted to.

Add the length parameter to the square() function:

```
def square(t, length):
    for i in range(0, 4):
        t.fd(length)
        t.lt(90)
```

Call the function:
- the function is expecting two arguments to be passed

```
square(frank, 300)
```

Again, we made the function more general, we can use it across more situations than we could before. Otherwise, if we didn't have this *generalized* function, we would have to define separate functions for each different square having varying sizes. This makes for more efficient function setup and function calling. Sure we could copy and paste; however, generalizing eliminates that extra step. Check out other situations by having each turtle you define draw out shapes in different sizes:

```
square(marie, 200)

square(albert, 50)
```

Summary

So got our first look into using turtle graphics with the python turtle module. We really got into some neat techniques, starting with creating a function. From there, we saw how we could reuse our function (which held code we were already reusing) by adding some parameters to generalize it; that is make it more useable across several situations. The more parameters added, the more we saw unique uses; concurrently, we saw how widely useful we could make our code.