Intro to Turtle graphics

Python Turtle graphics, inherited from the computer programming language Logo, gives us a graphics (and simple user interface) programming tool. From plotting data to creating unique pieces of art, we will make use of the graphical output in several problems this term.

The Python “Turtle” is essentially a robot that is controlled with Python code. The Turtle/robot comes equipped with multiple attributes, such as position, heading, color, and size.

We will be making use of the anonymous turtle for this project. This means you don’t need to set up the Turtle (t = turtle.Turtle() ) like in the book. Commands can be called directly, after using ‘from turtle import *’ (leave off the quotes). You need to import the turtle functions only once per Shell session. (But if the Shell restarts, you will need to execute ‘from turtle import *’ again.)

Try it yourself in the shell:
>>> from turtle import *

Turtle’s initial position is (0, 0) and its initial heading is 0 (East).

Make sure to try out the Turtle commands describe here in the shell to aid in understanding them.

**fd(distance) / forward(distance):** Move the Turtle forward by the specified distance, in the direction the Turtle is headed.

```
>>> fd(60)
```

Note how the command used is ‘fd(60)’ and not ‘turtle.fd(60)’. We are able to do this because we imported everything from the turtle module (from turtle import *). You will never need to preface any of your commands with turtle. (i.e. turtle.command() ) for this project.
**bk(distance) / back(distance) / backward(distance):** Move the Turtle backward by distance, opposite to the direction the Turtle is headed. Does not change the Turtle’s heading.

```
try it yourself in the shell:
>>> bk(60)
```

`fd` and `bk` are *relative* commands, moving the turtle `fd` or `bk` a certain distance from the turtle’s current position. There are also *absolute* commands, which move the turtle to a position on the (invisible) grid regardless of the turtle’s current location. The turtle canvas is an x, y grid, where (0, 0), the turtle’s initial position, is in the middle of the grid.

**setpos(x, y):** sets the new position for the Turtle. Lines get drawn between the old and new position, if the Turtle pen is down. Note that the new Turtle position is independent of its prior position, as opposed to `fd/bk` which move relative to the Turtle’s current position.

```
try it yourself in the shell:
>>> setpos(20, -20)
```

There are commands to tell us the turtle’s current position: `xcor()` returns the turtle’s position on the x-axis, `ycor()` returns the turtle’s location on the y-axis, and `position()` returns both.

```
try it yourself in the shell:
>>> setpos(-100, -50)
>>> xcor()
-100
>>> ycor()
-50
>>> position()
(-100.00,-50.00)
```
**reset():** Deletes the Turtle’s drawings from the screen, re-centers the Turtle and sets variables (pen size, speed, etc…) to their default values.

<table>
<thead>
<tr>
<th>Position of turtle: (20, -20)</th>
<th>Position of turtle: (0, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Try it yourself in the shell:
```python
>>> from turtle import *
>>> fd(20)
>>> rt(90)
>>> fd(20)
>>> reset()
```

**clear():** Deletes the Turtle’s drawings from the screen. Does not move or rotate the Turtle.

<table>
<thead>
<tr>
<th>Position of turtle: (20, -20)</th>
<th>Position of turtle: (20, -20)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Try it yourself in the shell:
```python
>>> from turtle import *
>>> fd(20)
>>> rt(90)
>>> fd(20)
>>> clear()
```

So far you’ve explored commands that change the turtle’s position. Other turtle commands affect the turtle’s heading:
**lt(angle) / left(angle):** Turn the Turtle left by angle units, *relative* to the Turtle’s current heading. (By default, the unit is degrees)

Try it yourself in the shell:
```python
>>> lt(90)
```

**rt(angle) / right(angle):** Turn the Turtle right by angle units, *relative* to the Turtle’s current heading.

Try it yourself in the shell:
```python
>>> rt(90)
```

Now combine them!
Look at the code below, try drawing what you think will happen, then execute the code to see:

```python
>>> fd(100)
>>> lt(90)
>>> bk(100)
>>> lt(90)
>>> fd(100)
>>> rt(270)
>>> bk(100)
```

What happens?

**setheading()** is an *absolute* turtle command. Try it yourself in the shell:
```python
>>> setheading(180)
```

and check the result with **heading()**:
```python
>>> heading()
```
More Turtle commands:

These Turtle commands are included in the Project 3 starter code. You can simply execute the starter code to use these commands, but you may find it interesting to explore them, too.

**title(titlestring):** sets title of Turtle window to titlestring.

<table>
<thead>
<tr>
<th>Python Turtle Graphics</th>
<th>Top of turtle window before a title is specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>title('Welcome to Computer Science at the UO!')</td>
<td>Specify a title</td>
</tr>
<tr>
<td>Welcome to Computer Science at the UO!</td>
<td>Top of turtle window after a title is specified</td>
</tr>
</tbody>
</table>

Try it yourself in the shell:

>>> from turtle import *
>>> title('Welcome to Computer Science at the UO!')

**speed(newspeed):** sets the Turtle’s speed to an integer value in the range 0 – 10. Strings can also be used to set the speed (see below).

- “fastest”: 0
- “fast”: 10
- “normal”: 6
- “slow”: 3
- “slowest”: 1

speed(“fast”) is the same as speed(10), speed(“slow”) is the same as speed(3), etc...

**pu() / penup():** Picks the Turtle pen up, so that if the Turtle moves, it will not draw.

Try it yourself in the shell:

>>> from turtle import *
>>> fd(30)
>>> pu()
>>> fd(30)
**pd() / pendown():** Puts the Turtle pen down, so if the Turtle moves, lines will be drawn.

<table>
<thead>
<tr>
<th>pd()</th>
<th>fd(20)</th>
<th>pd()</th>
</tr>
</thead>
<tbody>
<tr>
<td>pu()</td>
<td>fd(20)</td>
<td>fd(20)</td>
</tr>
</tbody>
</table>

Try it yourself in the shell:

```python
>>> from turtle import *
>>> fd(20)
>>> pu()
>>> fd(20)
>>> pd(20)
>>> fd(20)
```

**hideturtle():** Hides the Turtle arrow, so that only the marks the Turtle draws are seen.

<table>
<thead>
<tr>
<th>hideturtle()</th>
<th>fd(60)</th>
</tr>
</thead>
</table>

Try it yourself in the shell:

```python
>>> from turtle import *
>>> hideturtle()
>>> fd(60)
```
More Turtle commands:

`fillcolor('colorname')`: Sets the fill color for when the Turtle draws a shape.

```
fillcolor('blue')
```

`begin_fill()`: Tells Turtle that lines drawn after this command, are lines you plan to fill.

`end_fill()`: Tells Turtle to calculate the fill, considering all lines drawn since the last `begin_fill()` command.

Let’s try it!

Consider the following code, try figuring out what you think it will create, and then check it in the shell!

```python
>>> fillcolor('blue')
>>> begin_fill()
>>> fd(100)
>>> lt(120)
>>> fd(100)
>>> lt(120)
>>> fd(100)
>>> end_fill()
```

Were you right?

`stamp()`: Stamp a copy of the Turtle shape onto the canvas at the current Turtle position.

Try it yourself in the shell:

```python
>>> from turtle import *
>>> stamp()
>>> fd(60)
```
**dot()**: Puts a dot onto the canvas at the current Turtle position.

<table>
<thead>
<tr>
<th>Mark left by</th>
<th>dot()</th>
<th>fd(60)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Turtle</td>
</tr>
</tbody>
</table>

Note the similarity to *stamp*, the only difference is the shape being left.

Try it yourself in the shell:

```python
>>> from turtle import *
>>> dot()
>>> fd(60)
```

You can specify a size and color for the dot, for example:

```python
>>> dot(10, 'purple')
>>> fd(100)
```
**Python Notes for Project 3**

**for loop:** Often when writing code, you will find that you end up writing the same line multiple times over, this is when a for loop comes in handy. A for loop can be used to repeat the same line of code multiple times.

Here is an example of the type of for loop you will need to complete project 3:

```python
for i in range(4):
    print(i)
```

Try typing this for loop into your own shell and playing around with it. What happens when you change the number within range? What happens when you change ‘i’ to different variable names? Maybe try adding more prints, or other lines of code to run within the loop.

The following Python keywords/and syntax are included in the Project 3 starter code and will continue to be core elements of our Python programming:

**from turtle import *:** ‘from turtle’ means we are importing content from the turtle library. ‘*’ means we want to import everything in the library, so we can directly use it without having to add “turtle.” to the start of what we are using. i.e. `fd(100)` will make the turtle move (rather than `turtle.fd(100)`).

**NOTE:** An anonymous turtle is available after the turtle module is imported. We will use the anonymous turtle; you do not need and should NOT set up a named turtle (t = turtle.Turtle()) as shown in the text.

With the anonymous turtle, all of the turtle functions such as `fd`, `bk`, and so on can be used without referring to a specific turtle (for example, `fd`, not `t.fd`).

**def:** def is a Python keyword that marks the start of a function header. The pattern for creating a function being: `def`, function name, parameter list inside parentheses (though if no parameters are needed, the parenthesis are just left empty)

**Comments:** a comment is code that programmers can read when looking at a file, but is not executed by the Python interpreter.

# a hash mark like this indicates a single line comment
Multi-line comments are made by typing between two sets of triple quotes.

Double quotes (""") work as well – just make sure they are the same, before and after the comment.

**pass**: pass is a null operation, when it is executed nothing happens. It is generally used as a placeholder for where a statement is required by syntax, but no code is needed to be executed, or a programmer later wants to come back and write code at that place. Is a Python keyword.

**return**: indicates the end of the function. Is a Python keyword

**Assignment**: we can also assign values to variables using the ‘=’ operator.

Try these examples in the shell:

```python
>>> x = 45
>>> x
What is printed in the shell?

>>> course = ‘CIS122’
>>> course
What is printed in the shell?

>>> pi = 3.14
>>> pi
What is printed in the shell?
```

**Writing Functions**: for this project you will write your own functions. Functions generally have four main sections:

1. **the function header**
   ```python
def name(parameters if any):
   ```

2. **the docstring**
   ```
   ”””
   Information about the function/what the function does
   Example(s) of use – call function and show or describe expected result
   ”””
   ```

3. **the code within the function**
   ```python
   pass
   ```

4. **the return marking the end of the function**
   ```python
   return
   ```
**Function parameters:** In the past we have left the parentheses of our functions empty, we will now be putting variables into our parentheses. These are called function parameters. A function parameter allows you to pass a variable into a function when the function is called, this is called passing an argument to the function. As an example, let’s say we want a function that has the Turtle draw a line of a provided distance, and then return to where it started. How do you get the number for the distance into that function? You can use a parameter:

```python
def draw_line_and_return(distance):
    '''
    Draws a line the length of “distance” then returns the Turtle to its starting point
    >>> draw_line_and_return(40)
    '''
    fd(distance)
    pu()
    bk(distance)
    pd()
    return
```

Here our parameter is ‘distance’ note how instead of having a hardcoded number, ‘distance’ is used for fd and bk.

But what happens if someone calls the function without an argument (e.g. `draw_line_and_return()`)? Currently the function will just throw an error, but functions can also have **default parameters**.

A **default parameter** is a parameter that is set by the programmer, which a function can use if the user does not provide an argument.

For the previous example function, let’s change it so if the user calls the function without an argument, instead of having an error, 20 will be used:

```python
def draw_line_and_return(distance=20):
    '''
    Draws a line the length of “distance” then returns the Turtle to its starting point
    >>> draw_line_and_return(40)
    >>> draw_line_and_return()
    '''
    fd(distance)
    pu()
    bk(distance)
    pd()
    return
```

Note in the function header how instead of (distance) it is now (distance=20), the ‘=20’ is the key to setting the default value. What this says is that if a user doesn’t provide a value for ‘distance’, then ‘distance’ is, by default, set to 20.

Try it yourself, what happens if you change 20 to 60? What about other numbers?
More Turtle commands that may be of interest:

**screensize(canvaswidth, canvasheight):** If no arguments are given, returns the width and height of the canvas, else resizes the canvas the Turtle is drawing on.

![Default Python Turtle window](image1)

![screensize(1000,1000)](image2)

Canvas size has increased, scroll bars are now present

Try it yourself in the shell:
```python
>>> from turtle import *
>>> screensize(1000, 1000)
```

**bgpic():** Sets background image for the turtle canvas.
Note: image file must be in same folder as .py file. .jpeg files work best.

**pencolor(colorstring):** Sets the pen color. Note: once the pen color changes this only affects lines drawn after the color change occurred. Previously drawn lines will keep the same color that they were originally drawn in.

![fd(30) pencolor(“red”) fd(30)](image3)

Try it yourself in the shell:
```python
>>> from turtle import *
>>> fd(30)
>>> pencolor(“red”)
>>> fd(30)
```
**pensize(size)**: sets the size of the line the Turtle draws.

```
>>> from turtle import *
>>> fd(30)
pensize(3)
>>> fd(30)
```

Try it yourself in the shell:
Appendix

Turtle Commands
fd / forward ...... page 1
bk / back .............. page 2
xcor ................ page 2
ycor ................ page 2
position .............. page 2
setpos .............. page 2
reset .............. page 3
clear .............. page 3
lt / left .............. page 4
rt / right .............. page 4
setheading ........ page 4
heading ........ page 4
title ............ page 5
speed ............ page 5
pu / penup ........ page 5
pd / pendown .. page 6
hideturtle ........ page 6
fillcolor ........ page 7
begin_fill ........ page 7
end_fill ........ page 7
stamp ........ page 7
dot ........ page 8
screensize .... page 12
bgpic ........ page 12
 pencolor ........ page 12
pensize ........ page 13

Python
from turtle import * ..... page 1, 9
for loop ................ page 9
def ................ page 9
comments ................ page 9, 10
assignment ............. page 10
pass ................ page 10
return ................ page 10
functions ........ page 10
function parameters .... page 11
default parameters ..... page 11
Author: Brandon Bower Fall 2020
Updated by KFH Winter 2022