WELCOME
CIS 210 COMPUTER SCIENCE I

PROGRAMMING GUIDE – FALL 2020 WEEK 2
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 the anonymous turtle for this project. This means you don’t need to set up the Turtle \((t = \text{turtle.Turtle}())\) like in the book. Commands can be called directly, after using ‘from turtle import *’ (leave off the quotes).

**Turtle graphics commands you will need for Project 1a:**
The following Turtle commands are all you need for P1a. If a command is not listed in the project 1a section, do not use it for project 1a. Make sure to try out the commands 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.

<table>
<thead>
<tr>
<th><img src="image" alt="Turtle Forward Arrow" /></th>
<th>fd(60)</th>
<th><img src="image" alt="Turtle Forward Arrow" /></th>
</tr>
</thead>
</table>

Try it yourself in the shell:

```python
>>> from turtle import *
>>> 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 \((\text{from turtle import *})\). You will never need to preface any of your commands with turtle. \((\text{i.e. turtle.command()})\) 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:
```python
>>> from turtle import *
>>> bk(60)
```

**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
>>> from turtle import *
>>> lt(90)
```

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

Try it yourself in the shell:
```python
>>> from turtle import *
>>> rt(90)
```
Now combine them!
Look at the code below, try drawing what you think will happen, then execute the code to see:

```python
>>> from turtle import *
>>> fd(100)
>>> lt(90)
>>> bk(100)
>>> lt(90)
>>> fd(100)
>>> rt(270)
>>> bk(100)
```

What happens?

**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. 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)
```
More Turtle commands:
These Turtle commands are included in the Project 1a starter code. You can simply execute the starter code to use these commands, but you may find it interesting to explore them, too.

**reset()**: Deletes the Turtle’s drawings from the screen, re-centers the Turtle and sets variables (pen size, speed, etc...) to their default values.

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.

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

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

Try it yourself in the shell:
```python
>>> 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...

**bgpic():** Sets background image.
Note: image file must be in same folder as .py file.

** 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.

Try it yourself in the shell:
>>> from turtle import *
>>> fd(30)
>>> pencolor(“red”)  
>>> fd(30)
**screensize(canvaswidth, canvashight):** If no arguments are given, returns the width and height of the canvas, else resizes the canvas the Turtle is drawing on.

Try it yourself in the shell:
>>> from turtle import *
>>> screensize(1000, 1000)
Python
These Python keywords/and syntax are included as part of the 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, as opposed to having to type Turtle.fd(100)
from and import are Python keywords

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 there are 3 before, and 3 after.
"""

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
(1) The file header comments provide information about this Python file (program):

```
... Title: a one-line description/title for the program
Author: Your name
Credit: reference any other sources (materials, people, etc.) for this work
...''
```

(2) The import statement provides access to turtle module functions when the function is executing.
import is a Python keyword. By convention, import statements appear at the top of the program file,
after the file header and before the rest of the code.
```
from turtle import *
```

(3) Note the structure of the Python `uo_guide` function:

```
def uo_guide_start():
    ""
    Welcome to the UO! Welcome to Computer Science!
    Guide students from the EMU Lawn to Dechutes Hall, home of the
    Computer Science Department, and then to Price Science Commons
    (Science Library), home of B004/A computer lab and study space.
    >>> uo_guide_start()
    ""
    # setting the scene (supply this code)
    reset()
    clear()
    title('Welcome to Computer Science at the UO!')
    color('purple')
    pensize(3)
    speed('slowest')
    bgpic('uo-campus_map.png')
    screensize(1195, 691)
    stamp() #mark start of route on EMU East lawn
    # replace pass with your code
    # guide to Dechutes
    pass
    # guide to Price Science Commons
    pass
    return
```

---

function header: keyword `def`, function name, parameter list inside parentheses (empty here, `aso_guide` has no parameters)

function docstring: inside triple quotes – a brief description of the function followed by an
example call to the function

function code: some is provided for you; you will write the rest

return statement: indicates the end of the function. `return` is a Python keyword.
Project 1b “Art Show”

You’ve seen that Turtle can be used to create maps to aid others in the finding of specific locations. What else can Turtle be used for, though? What about in the creation of art?

**Turtle graphics**

**Turtle graphics commands you will need for Project 1b:**

The following Turtle commands, as well as the commands we learned in P1a, are all you need for P1b. If a command is not listed in the P1a or P1b section, do not use it for this project.

Make sure to use the shell to aid in understanding the commands.

We will once again use the anonymous turtle, as well as directly using commands (i.e. just call command(), not turtle.comand() )

**fillcolor('colorname')**: Sets the fill color for when the Turtle draws a shape. This may seem like the pencolor command, but pencolor changes the color of the lines drawn, where fillcolor decides the color of the negative space between the drawn lines.

**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
>>> from turtle import *
>>> fillcolor('blue')
>>> begin_fill()
>>> fd(100)
>>> lt(120)
>>> fd(100)
>>> lt(120)
>>> fd(100)
>>> end_fill()
```

Were you right?
**More Turtle commands:**
These Turtle commands are included in the Project 1b starter code. You can simply execute the starter code to use these commands, but you may find it interesting to explore them, too.

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

```python
>>> from turtle import *
>>> fd(30)
>>> pu()
>>> fd(30)
```

**pd() / pendown():** Puts the Turtle pen down, so if the Turtle moves, lines will be drawn.

```python
>>> from turtle import *
>>> fd(20)
>>> pu()
>>> fd(20)
>>> pd()
>>> fd(20)
```
**hideTurtle():** Hides the Turtle arrow, so that only the marks the Turtle draws are seen.

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

**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.

```python
>>> from turtle import *
>>> setpos(20, -20)
```

**pensize(size):** sets the size of the line the Turtle draws.

```python
>>> fd(30)
>>> pensize(3)
>>> fd(30)
```
Python

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 over.

Here is an example of the type of for loop you will need to complete project 1b:

```
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.

Assignment: we saw the use of the variable ‘i’ in the above for loop, but we can also assign values to variables outside of for loops using the ‘=’ operator. For example, `x = 45` or `class = ‘CIS210’`

Try these examples in the shell:
```
>>> x = 45
>>> x
```

What is printed in the shell?
```
>>> class = ‘CIS210’
>>> class
```

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. Start by looking at the functions already provided to you with the project starter code from P1a as well as P1b, and then try writing your own with a similar structure, but with different contents. Functions generally have 4 main sections:

1 the function header
   def name(parameters if any, will be empty for this assignment):
2 the docstring
   ""
   Information about the function/what the function does
   Examples of use
   ""
3 the code within the function
   pass
4 the return marking the end of the function
   return

Project 1c “Art Show Better”
We've made some great artwork with “Art Show”, but the program really can only make one picture, and that picture is hard to tweak. Function Parameters can be used to make our picture easier to modify.

Turtle graphics
Turtle graphics commands you will need for Project 1c:
There are no new commands, we will be using the same commands we learned in P1a, and P1b. If a command is not listed in the P1a or P1b section, do not use it for this project.
We will once again use the anonymous turtle, as well as directly using commands (i.e. just call command(), not turtle.comand() )

Python
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:

   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?
Can you have multiple default parameters?
Project 2a “Mars Rover”
Time for something new with Turtle, exploring space! Or specifically, the surface of Mars. We will learn more absolute ways to move our Turtle, as opposed to the relative ways we were using before.

Turtle graphics
Turtle graphics commands you will need for Project 1b:
The following Turtle commands, as well as the commands we learned in P1a, and P1b are all you need for P2a. If a command is not listed in the P1a, P1b or P2a section, do not use it for this project.
Make sure to use the shell to aid in understanding the commands.
We will once again use the anonymous turtle, as well as directly using commands (i.e. just call command(), not turtle.command() )

**seth(to_angle) / setheading(to_angle):** Sets the orientation of the Turtle to to_angle. to_angle by default is in degrees. The current angle of the Turtle is not considered when changing to to_angle. The diagram below shows which direction the Turtle will face, given specific to_angle inputs.

```
>>> from turtle import *
>>> seth(45)
>>> seth(-45)
>>> seth(0)
```

Watch the Turtle move after each command, try other angles as well!

towards(x, y): returns a number, that when used with seth / setheading, will point the Turtle towards x, y.

```
heading = towards(20, -20)
seth(heading)
```

*Or*

```
seth(towards(20, -20))
```

Try it yourself in the shell:
```
>>> from turtle import *
>>> heading = towards(20, -20)
>>> seth(heading)
```
**write(text):** Writes ‘text’ string at current location of the Turtle. Note: angle of Turtle does not influence the angle at which the text is written, also write will work regardless of if the pen is down. (Write is for the optional portion, and is not required to complete project 2a)

Try it yourself in the shell:
```python
>>> from turtle import *
>>> write("hello world")
```

Try rotating and moving the Turtle between writes as well, and see what happens!

**Python**

**print(thing_to_print):** prints whatever is in the parenthesis, to the shell. Accepts all variable types, if multiple items are desired to be printed, they can be comma separated.

Some examples to try in the shell:
```python
>>> x = 5
>>> print(x)

>>> x = 'hello'
>>> y = 'cis'
>>> z = 210
>>> print(x, y, z)

>>> pi = 3.14
>>> print(pi)
```

Special characters can also be used with print. For special characters, the escape character ‘\’ tells python the letter immediately following will have a special use. The letters that don’t immediately follow, though, will be treated as normal characters. For example, if you have ‘\na’ only the ‘n’ will be ‘escaped’ and considered for its special character use, the ‘a’ will be treated as a normal character.

Examples:

\t' creates a tab
```python
>>> print("\ttabbed text")
```

\n' creates a new line
```python
>>> print("line\nnew line")
```
**Return Values:** Up until now we have been using return only to mark the end of a function, but return also has another, generally more important, use: passing values from function to function. We can return a value simply by placing if after the return at the end of the function:

```python
def mult_60(number):
    '''
    Multiplies some number by 60
    >>> mult_60(1)
    60
    '''
    result = number * 60
    return result
```

*Note in the example of use how there is now another line showing the value that is returned (4 here)*

This can be powerful for combining functions. Let’s say we want to figure out how many seconds are in a day. We can use `mult_60` to figure out how many seconds are in an hour, but we could write another function that multiplies by 24 to then figure out how many seconds are in a day:

```python
def mult_24(number):
    '''
    Multiplies some number by 24
    >>> mult_24(2)
    48
    '''
    result = number * 24
    return result
```

Now we can write a main function to call both previously written functions to figure out how many seconds are in a day.

```python
def main():
    '''Calls `mult_60` and `mult_24` to calculate how many seconds are in a day'''
    SECS_PER_MIN = 60
    secs_per_hour = mult_60(SECS_PER_MIN)
    secs_per_day = mult_24(secs_per_hour)
    print(secs_per_day)
    return
```

You may notice that the ‘SECS_PER_MIN’ variable is in all capitals, this is because it is a **Python Constant**. While Python does not support constant values for variables, by convention python programmers communicate that a variable’s value is not meant to change by giving the variable an uppercase name and defining it at the start of function code. Since there will always be 60 seconds in a minute (unless scientists decide to go and change it for some reason) it is declared as a constant. ‘secs_per_hour’ and ‘secs_per_day’ are not declared as constants as their values are not known before the program runs, they rely on the return values of functions, and thus could in theory change with different executions of the program (though in this case they will not).
Now, what happens when ‘main’ is executed?
What would happen if we changed mult_60 so that it printed its result instead of returning it?

def mult_60(number):
    
    Multiplies some number by 60
    >>> mult_60(1)
    60
    
    result = number * 60
    print(result)
    return number

Would this change what happens when main is executed? What would happen?

Project 2b “Variations” / Project 2c “Approximate Square Root”
We are finished with Turtle for now, but there are still lots of neat things to be learned about for Python itself. In fact, Turtle isn’t the only module we can import and make use of.

The Math Module, for example, is one such module that we can also use. An aside about modules, we have used ‘from module import *’, and this imports all aspects of a module so they can be used directly. But there are other ways to access parts of a module. If we only need a specific function, let’s say the square root function, we can use a similar import statement, but swap the ‘*’ for the name of the function we need. So instead of ‘from math import *’ we specify sqrt: ‘from math import sqrt’ now sqrt can be used directly. Give it a try.

>>> from math import sqrt
>>> sqrt(4)

Let’s say we want multiple functions, too many to import them by name, but not enough to justify importing the entire module using ‘from module import *’. In this case we just use: ‘import module’ and then we can access the functions of the module using ‘module.function()’.

Here is how to access sqrt this way:

>>> import math
>>> math.sqrt(4)

sqrt is just one example of the helpful tools the math module can provide.

round(some_float): round takes in a floating-point number and rounds it to the nearest integer. This works as expected, except for one specific instance.

Consider the following:

>>> round(3.5)
>>> round(4.5)

What do you expect to get for each? What do you get when tried in the shell?

You probably expected to get 4 and 5 respectively, but instead got 4 and 4. This is due to the fact that Python utilizes banker’s rounding, which states that if a number is halfway between two integers, then round to the nearest even integer.
**Modulo, %:** Modulo is a binary operator (operates on two numbers), represented in Python with the percentage ‘%’ symbol, that tells the remainder of the first number divided by the second. For example, how many times does 4 go into 9? 2 times (which is 8), with 1 remaining (9 – 8 = 1). Thus 9 % 4 will return 1. This can be helpful in a lot of different ways; one is deciding if a number is even. An even number mod (modulus) 2 will always return 0, and an odd number mod 2 will always return 1.

Note if the number on the right side is larger, the number on the left will always be returned. For example, 2 % 8 will just return 2, as 8 goes into 2 zero times, with a remainder of 2.

**Integer Division, //:** Integer division is another binary operator, represented in Python with double forward slashes ‘//’, that is sort of the compliment to modulo. Integer division tells you how many times one number evenly goes into another number. For example, remember how 9 % 4 = 1? Well 9 // 4 = 2, as 4 goes into 9 two times. Integer division is useful for “how many x in y” sort of questions. For example, if I have 221 seconds, how many full minutes do I have?

Let’s find out:
>>> 221 // 60

Note If the number on the right is larger the return value will always be 0, as a larger number can never go evenly into a smaller number, e.g. 2 // 9 = 0.

**abs(number):** abs, or absolute value, is exactly what you would expect, it does in Python the same as it does in math: ensures a number is positive. For example abs(-4) will return 4, abs(4) will also return 4. Try it for yourself in the shell:
>>> abs(-4)
>>> abs(4)
>>> abs(0)

Works with floats as well:
>>> abs(-3.14)
Turtle Commands

- `fd` / `forward` .......... project 1a, page 1
- `bk` / `back` .............. project 1a, page 2
- `lt` / `left` ............... project 1a, page 2
- `rt` / `right` ............. project 1a, page 2
- `stamp` ................... project 1a, page 3
- `dot` ...................... project 1a, page 3
- `reset` .................... project 1a, page 4
- `title` .................... project 1a, page 4
- `speed` .................... project 1a, page 5
- `bgpic` .................... project 1a, page 5
- `pencolor` ................. project 1a, page 5
- `screensize` .............. project 1a, page 6
- `fillcolor` ............... project 1b, page 9
- `begin_fill` ............. project 1b, page 9
- `end_fill` ............... project 1b, page 9
- `pu` / `penup` .......... project 1b, page 10
- `pd` / `pendown` ....... project 1b, page 10
- `hideTurtle` ............. project 1b, page 11
- `setpos` ................... project 1b, page 11
- `pensize` ................. project 1b, page 11
- `seth` / `setheading` .. project 2a, page 15
- `towards` ................. project 2a, page 15
- `write` .................... project 2a, page 16

Python

- `from turtle import *` ... project 1a, page 7
- `def` ....................... project 1a, page 7
- `comments` ............... project 1a, page 7
- `pass` ...................... project 1a, page 7
- `return` ................... project 1a, page 7
- `for loop` ................ project 1b, page 12
- `assignment` ............. project 1b, page 12
- `functions` ............... project 1b, page 13
- `function parameters` .. project 1c, page 13
- `default parameters` .... project 1c, page 14
- `print` ..................... project 2a, page 16
- `return values` .......... project 2a, page 17
- `Python constant` ....... project 2a, page 18
- `Math Module` .......... project 2b, page 18
- `round` ..................... project 2b, page 18
- `Modulo` .................. project 2b, page 19
- `Integer Division` ........ project 2b, page 19
- `abs` ....................... project 2c, page 19

Author: Brandon Bower Fall 2020