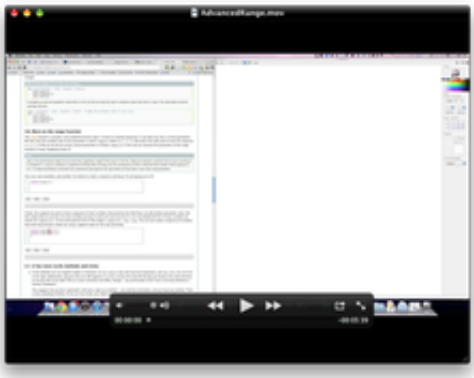
The range Function



In our simple example from the last section (shown again below), we used a list of four integers to cause the iteration to happen four times. We said that we could have used any four values. In fact, we even used four colors.

```
import turtle            #set up alex
wn = turtle.Screen()
alex = turtle.Turtle()

for i in [0,1,2,3]:      #repeat four times
    alex.forward(50)
    alex.left(90)

wn.exitonclick()
```

It turns out that generating lists with a specific number of integers is a very common thing to do, especially when you want to write simple `for loop` controlled iteration. Even though you can use any four items, or any four integers for that matter, the conventional thing to do is to use a list of integers starting with 0. In fact, these lists are so popular that Python gives us special built-in `range` objects that can deliver a sequence of values to the `for` loop. The sequence provided by `range` always starts with 0. If you ask for `range(4)` , then you will get 4 values starting with 0. In other words, 0, 1, 2, and finally 3. Notice that 4 is not included since we started with 0. Likewise, `range(10)` provides 10 values, 0 through 9.

```
for i in range(4):
    # Executes the body with i = 0, then 1, then 2, then 3
for x in range(10):
    # sets x to each of ... [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note

Computer scientists like to count from 0!

So to repeat something four times, a good Python programmer would do this:

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

The range (http://docs.python.org/py3k/library/functions.html?highlight=range#range) function is actually a very powerful function when it comes to creating sequences of integers. It can take one, two, or three parameters. We have seen the simplest case of one parameter such as `range(4)` which creates `[0, 1, 2, 3]`. But what if we really want to have the sequence `[1, 2, 3, 4]`? We can do this by using a two parameter version of `range` where the first parameter is the starting point and the second parameter is the ending point. The evaluation of `range(1,5)` produces the desired sequence. What happened to the 5? In this case we interpret the parameters of the range function to mean range(start,stop+1).

Note

Why in the world would range not just work like range(start, stop)? Think about it like this. Because computer scientists like to start counting at 0 instead of 1, `range(N)` produces a sequence of things that is N long, but the consequence of this is that the final number of the sequence is N-1. In the case of start, stop it helps to simply think that the sequence begins with start and continues as long as the number is less than stop.

Here are a two examples for you to run. Try them and then add another line below to create a sequence starting at 10 and going up to 20 (including 20).

```
1 print(range(4))
2 print(range(1,5))
3
```

ActiveCode: 1 (ch03_5)

Run

```
[0, 1, 2, 3]
[1, 2, 3, 4]
```

Codelens will help us to further understand the way range works. In this case, the variable `i` will take on values produced by the `range` function.

```
1  for i in range(10):
2      print(i)
```

<< First | < Back | Program terminated | Forward > | Last >>

line that has just executed

next line to execute

Program output:

```
0
1
2
3
4
5
6
7
8
9
```

Frames Objects

Global variables

i	9

CodeLens: 1 (rangeme)

Finally, suppose we want to have a sequence of even numbers. How would we do that? Easy, we add another parameter, a step, that tells range what to count by. For even numbers we want to start at 0 and count by 2's. So if we wanted the first 10 even numbers we would use `range(0,19,2)`. The most general form of the range is `range(start, stop, step)`. You can also create a sequence of numbers that starts big and gets smaller by using a negative value for the step parameter.

```
1 print(range(0,17,2))
2 print(range(0,20,2))
3 print(range(10,0,-1))
4
```

ActiveCode: 2 (ch03_6)

Run

```
[0, 2, 4, 6, 8, 10, 12, 14, 16]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Try it in codelens.

```
1  for i in range(0,20,2):
2      print(i)
```

<< First | < Back | Program terminated | Forward > | Last >>

line that has just executed

next line to execute

Program output:

```
0
2
4
6
8
10
12
14
16
18
```

Frames Objects

```
Global variables
          i  18
```

CodeLens: 2 (rangeme2)

Check your understanding

trl-20: In the command range(3, 10, 2), what does the second argument (10) specify?

- (•) a) Range should generate a list that stops at 9 (including 9).
- ( ) b) Range should generate a list that starts at 10 (including 10).
- ( ) c) Range should generate a list starting at 3 that stops at 10 (including 10).
- ( ) d) Range should generate a list using every 10th number between the start and the stopping number.

Check Me | Compare Me

Correct!! Range will generate the list [3, 5, 7, 9].

trl-21: What command correctly generates the list [2, 5, 8]?

- ( ) a) range(2, 5, 8)
- ( ) b) range(2, 8, 3)
- (•) c) range(2, 10, 3)
- ( ) d) range(8, 1, -3)

Check Me | Compare Me

Correct!! The first number is the starting point, the second is the maximum allowed, and the third is the amount to increment by.

trl-22: What happens if you give range only one argument? For example: range(4)

(•)

a) It will generate a list starting at 0, with every number included up to but not including the argument it was passed.

b) It will generate a list starting at 1, with every number up to but not including the argument it was passed.

c) It will generate a list starting at 1, with every number including the argument it was passed.

d) It will cause an error: range always takes exactly 3 arguments.

Check Me | Compare Me

Correct!! Yes, if you only give one number to range it starts with 0 and ends before the number specified incrementing by 1.

A Few More turtle Methods and Observations

Here are a few more things that you might find useful as you write programs that use turtles.

- Turtle methods can use negative angles or distances. So `tess.foward(-100)` will move tess backwards, and `tess.left(-30)` turns her to the right. Additionally, because there are 360 degrees in a circle, turning 30 to the left will leave you facing in the same direction as turning 330 to the right! (The on-screen animation will differ, though — you will be able to tell if tess is turning clockwise or counter-clockwise!)

  This suggests that we don't need both a left and a right turn method — we could be minimalists, and just have one method. There is also a *backward* method. (If you are very nerdy, you might enjoy saying `alex.backward(-100)` to move alex forward!)

  Part of *thinking like a scientist* is to understand more of the structure and rich relationships in your field. So revising a few basic facts about geometry and number lines, like we've done here is a good start if we're going to play with turtles.

- A turtle's pen can be picked up or put down. This allows us to move a turtle to a different place without drawing a line. The methods are `up` and `down`. Note that the methods `penup` and `pendown` do the same thing.

```
alex.up()
alex.forward(100)     # this moves alex, but no line is drawn
alex.down()
```

- Every turtle can have its own shape. The ones available "out of the box" are `arrow`, `blank`, `circle`, `classic`, `square`, `triangle`, `turtle`.

```
...
alex.shape("turtle")
...
```

- You can speed up or slow down the turtle's animation speed. (Animation controls how quickly the turtle turns and moves forward). Speed settings can be set between 1 (slowest) to 10 (fastest). But if you set the speed to 0, it has a special meaning — turn off animation and go as fast as possible.

```
alex.speed(10)
```

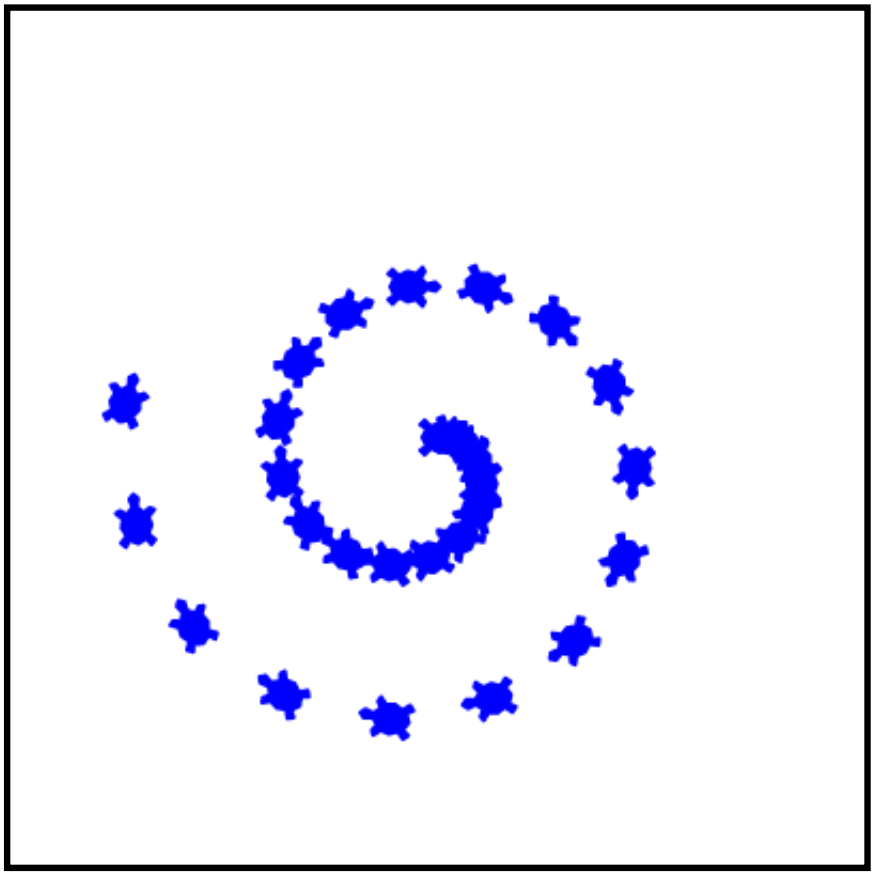
- A turtle can "stamp" its footprint onto the canvas, and this will remain after the turtle has moved somewhere else. Stamping works even when the pen is up.

Let's do an example that shows off some of these new features.

```
tess = turtle.Turtle()
tess.color("blue")
tess.shape("turtle")

print(range(5,60,2))
tess.up()                       # this is new
for size in range(5,60,2):      # start with size = 5 and grow by 2
    tess.stamp()                # leave an impression on the canvas
    tess.forward(size)          # move tess along
    tess.right(24)              # and turn her

tess.color("white")

wn.exitonclick()

```

ActiveCode: 3 (ch03_7)

Run



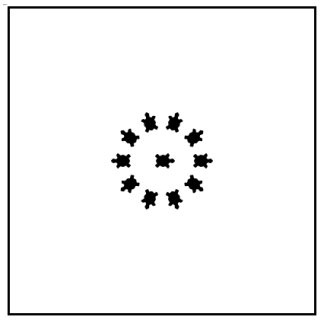
```
[5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49, 51,
53, 55, 57, 59]
```

The list of integers shown above is created by printing the `range(5,60,2)` result. It is only done to show you the distances being used to move the turtle forward. The actual use appears as part of the `for` loop.

One more thing to be careful about. All except one of the shapes you see on the screen here are footprints created by `stamp` . But the program still only has *one* turtle instance — can you figure out which one is the real tess? (Hint: if you're not sure, write a new line of code after the `for` loop to change tess' color, or to put her pen down and draw a line, or to change her shape, etc.)

Mixed up program

trl-23: The following program uses the stamp method to create a circle of turtle shapes as shown to the left, but the lines are mixed up. The program should do all necessary set-up, create the turtle, set the shape to "turtle", and pick up the pen. Then the turtle should repeat the following ten times: go forward 50 pixels, leave a copy of the turtle at the current position, reverse for 50 pixels, and then turn right 36 degrees. After the loop, set the window to close when the user clicks in it.



Drag the blocks of statements from the left column to the right column and put them in the right order with the correct indention. Click on *Check Me* to see if you are right. You will be told if any of the lines are in the wrong order or are incorrectly indented.

Drag from here

Drop blocks here

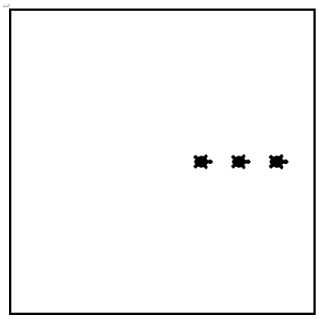
```
import turtle
wn = turtle.Screen()
jose = turtle.Turtle()
jose.shape("turtle")
jose.penup()
for size in range(10):
    jose.forward(50)
    jose.stamp()
    jose.forward(-50)
    jose.right(36)
wn.exitonclick()
```

Check Me Reset

Perfect!

Mixed up program

trl-24: The following program uses the stamp method to create a line of turtle shapes as shown to the left,



but the lines are mixed up. The program should do all necessary set-up, create the turtle, set the shape to "turtle", and pick up the pen. Then the turtle should repeat the following three times: go forward 50 pixels and leave a copy of the turtle at the current position. After the loop, set the window to close when the user clicks in it.

Drag the blocks of statements from the left column to the right column and put them in the right order with the correct indention. Click on *Check Me* to see if you are right. You will be told if any of the lines are in the wrong order or are incorrectly indented.

Drag from here

Drop blocks here

```
import turtle
wn = turtle.Screen()
```

```
nikea = turtle.Turtle()
```

```
nikea.shape("turtle")
```

```
nikea.penup()
```

```
for size in range(3):
```

```
    nikea.forward(50)
```

```
    nikea.stamp()
```

```
wn.exitonclick()
```

Check Me | Reset

Perfect!

Lab

- Turtle Race (http://dcs.asu.edu/faculty/abansal/CST100/Labs/TurtleGraphics-TurtleRacingLab.html) In this guided lab exercise we will work through a simple problem solving exercise related to having some turtles race.

Summary of Turtle Methods

Method	Parameters	Description
Turtle	None	Creates and returns a new turtle object
forward	distance	Moves the turtle forward
backward	distance	Moves the turle backward
right	angle	Turns the turtle clockwise
left	angle	Turns the turtle counter clockwise
up	None	Picks up the turtles tail
down	None	Puts down the turtles tail
color	color name	Changes the color of the turtle's tail
fillcolor	color name	Changes the color of the turtle will use to fill a polygon
heading	None	Returns the current heading
position	None	Returns the current position
goto	x,y	Move the turtle to position x,y
begin_fill	None	Remember the starting point for a filled polygon
end_fill	None	Close the polygon and fill with the current fill color
dot	None	Leave a dot at the current position
stamp	None	Leaves an impression of a turtle shape at the current location
shape	shapename	Should be 'arrow', 'classic', 'turtle', or 'circle'

shape shapename Should be 'arrow', 'classic', 'turtle', or 'circle'

Once you are comfortable with the basics of turtle graphics you can read about even more options on the Python Docs Website (http://docs.python.org/dev/py3k/library/turtle.html). Note that we will describe Python Docs in more detail in the next chapter.

Note

This workspace is provided for your convenience. You can use this activecode window to try out anything you like.

```
1
2
```

ActiveCode: 4 (scratch_03)

Run