Python Lesson 2

For Loops

In the last lesson we saw how the `while` statement can be used to create loops. Loops are blocks of code which are repeated many times.

There is another type of loop in Python, a `for` loop. This is used for processing items in a **sequence** one at a time.

The syntax of a `for` loop is "`for` *`<variable name>`* `in` *`<sequence>`*`:`". Each time through the loop the next value from the sequence will be assigned to the variable.

Here's something you can try in the IDLE Shell:

```
for name in ["Jennifer", "Alison", "Phillipa", "Sue"]:
    print("I wrote this song for you, " + name + ".")
```

```
I wrote this song for you, Jennifer.
I wrote this song for you, Alison.
I wrote this song for you, Phillipa.
I wrote this song for you, Sue.
```

For loops work by looping over any types of sequence, but what *exactly* is a sequence?

Sequences

You've already seen a couple of different types of sequence although you may not know it!

- **Lists** are sequences of **values**.
- **Strings** are sequences of **characters** (letters, digits, punctuation, other symbols). You can process the characters in a string one at a time with a `for` loop.

  ```
  for letter in "AEIOU":
      number=ord(letter)-ord("A")+1
      print(letter, "is letter", number, "in the alphabet")
  ```

  ```
  A is letter 1 in the alphabet
  E is letter 5 in the alphabet
  I is letter 9 in the alphabet
  O is letter 15 in the alphabet
  U is letter 21 in the alphabet
  ```

- **Ranges** are type of sequence returned by the `range()` built-in function.

```
for number in range(0, 10):
    print(number, end=" ")
```

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

You can see in this example that the last item in the range is not included. The sequence returned by `range(a, b)` contains items that are *greater than or equal to* `a`, and *less than* `b`.

Getting this wrong is a common source of bugs in Python programs! It is so common it has even got a name: an "off-by-one" error.

- **Enumerations** are type of sequence which includes both a value and the number of the value in the sequence. You can create an enumeration from an existing sequence using the `enumerate()` function.

```
for index, letter in enumerate("AEIOU", 1):
    print(index, letter)
```

```
1 A
2 E
3 I
4 O
5 U
```

Turtle Graphics

Python includes a module called 'turtle' for drawing shapes. We can use it to demonstrate `for` loops.

The Python way of looping 4 times is to use a `for` loop with `range(0, 4)`.

Create a new program file called shapes.py. You can edit this file to create a nice picture.

```
import turtle

for side in range(0, 4):
    turtle.forward(200) # pixels
    turtle.right(90)    # degrees
```

Run this program. What shape does this draw? Can you modify the program to draw an octagon (a shape with eight sides)?

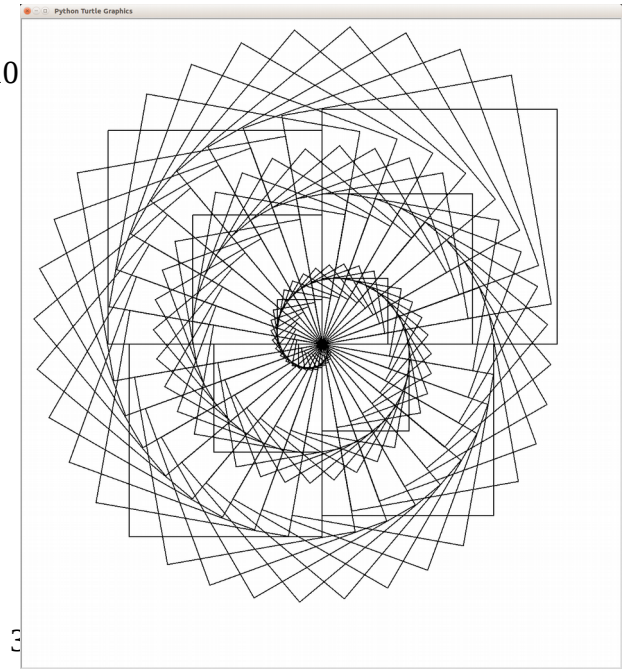
You can nest `for` loops inside each other to draw lots of shapes. See what this draws.

```
import turtle

turtle.speed("fastest") # don't take too long!
turtle.pensize(2)       # use a thicker pen
turtle.pencolor("red")  # we can pick a colour – note the US spelling!

for square in range(0, 10):
    for side in range(0, 4):
        turtle.forward(200)
        turtle.right(90)
    turtle.right(36)
```

Can you edit the program to draw a shape that looks a bit like this? It doesn't need to be exactly the same, you can be creative!

This picture is drawing 90 squares each rotated 10 degrees from each other.

The size of each square starts small (5 pixels on each side) and gets bigger by 5 pixels each time.



Functions

We've used a number of built-in Python **functions** like print() and range(), and some functions provided by modules like random.choice() and turtle.forward(). Now it's time to start creating our own functions!

User defined functions are a powerful features of most programming languages (but missing in Scratch!) which allow you to avoid repeating yourself all the time. If you find yourself performing the same lines of code again and again in a program consider creating a function that says how to perform the task once, then call the function whenever you need it.

Functions also help break your program up into smaller, more manageable sized pieces which are easier to understand.

You can create your own function for drawing a square like this:

```
def draw_square(size):
    for side in range(0, 4):
        turtle.forward(size)
        turtle.right(90)
```

This defines (using the `def` keyword) a new function called `draw_square` which expects a single **argument** called `size`. Function arguments are variables which will be set when someone uses the function.

Now the rest of the program can draw a square with just one line of code:

```
draw_square(100)
```

This will cause the block of code in the `draw_square(size)` function to be run with its `size` variable set to the value 100.

Let's see how using a function simplifies the previous example.

```
for square in range(0, 10):
    draw_square(200)
    turtle.right(36)
```

We've managed to get rid of the nested `for` loops which makes the program easier to read.

Exercise: Can you create a function `draw_shape(sides, size)` which can draw any regular polygon, not just squares? The sides argument will be the number of sides that the shape should have.

Test your function by drawing triangles, squares and pentagons!

Can you modify the draw_square(size) function to use draw_shape() instead of using its own loop? Functions can call other functions.

Returning Values from Functions

Functions can return values to the caller. For example, the `random.choice(list)` function returns a random item from the list.

Your own functions can return values too. To do this, use the `return` statement.

```
def is_even(value):
    if value % 2 == 0:
        return True
    else:
        return False

print(is_even(5)) # prints False
print(is_even(6)) # prints True
```

The `%` (percent) operator here is called "**modulo**", but it really means "the remainder from division". "`value % 2`" divides value by 2 and returns the remainder.

The `if` statement says that if the remainder when dividing by two equals zero, which means that the value was even, then we should return the boolean value `True`.

Variables in Functions

This section may be too much information! Don't worry if you don't understand it straight away.

Function arguments and other variables created inside a function are a bit special. They are called *local variables*.

This means that these variables and their values are completely hidden inside the function. The variables will be destroyed when the function finishes. Code outside of the function cannot see them. You can even have variables inside the function whose name is the same as a variable outside the function.

```
def greet(person):
    message="Hello " + person
    print(message)

message="Are we nearly there yet?"
greet("Chris")
print(message)
```

This prints:

```
Hello Chris
Are we nearly there yet?
```

Although the `greet()` function sets a variable called `message`, this `message` variable is different to the one with the same name used outside the function.

We say that the *scope* of the message variable is *local* to the function.

Local variables are a very good thing, because they mean that you don't need to worry (much!) about variable name conflicts between different parts of your program, or with modules written by other programmers. Functions can be simple isolated pieces of code.

Variables defined outside a function are called *global variables*. These variables can be read anywhere, including inside functions. In this example `greeting_prefix` is a global variable.

```
greeting_prefix="Hello"

def greet(person):
    message=greeting_prefix + " " + person
    print(message)

greet("Chris")
print(greeting_prefix)
```

Creating or modifying a global variable from inside a function is normally a bad idea, but it is possible. You need to explicitly declare the scope of the variable as global inside the function using the `global` keyword.

```
greeting_prefix="Hello"

def change_greeting_prefix():
    # Without the next line this function would be creating a new local
    # variable called greeting_prefix instead of modifying the global.
    global greeting_prefix
    greeting_prefix="Howdy"

change_greeting_prefix()
print(greeting_prefix) # prints "Howdy"
```

Try to avoid modifying global variables from within functions. If you modify global variables from lots of different functions is can be hard to understand why the value of a variable has changed unexpectedly.

It's normally a better idea for your function to return a result value to the caller with a `return` statement than to write the result value into a global variable.