Episode 4
Loops and Conditionals

Ultimately, a computer cannot do very much. It can store to memory, fetch from memory, perform a limited set of computations and comparisons, and make decisions. But those simple operations are used to create all the many computer programs we use every day.

Conditionals

We'll start with decisions. Decisions are made when some condition is either *true* or *false*. A variable that can take only the two special Python values `True` and `False` is called a Boolean variable. Expressions that evaluate to `True` or `False` are Boolean expressions. Comparisons are a common way to form Boolean expressions. Numerical types and strings have comparison operators defined on them. Numbers can be compared as less than, less than or equal, greater than, greater than or equal, equal, and not equal. The corresponding operators are `<`, `<=`, `>`, `>=`, `==`, and `!=`. Numbers are compared based on standard arithmetical ordering. The same operators can be used with strings, but for strings we use lexical ordering, a standard defined as part of the character set. For instance, capital letters are "less than" lower-case letters, i.e. `'Z'<'a'`.

We can use comparisons directly or we can combine them with the Boolean oporators `and`, `or`, and `not`. The `and` and `not` operators are just what you would expect, but `or` may be a little unusual. In ordinary language, or is generally exclusive. You can have cake or ice cream but not both. In most programming languages, `or` is inclusive.

```
cake_ok=True
ice_cream_ok=True
cake_or or ice_cream_ok # True
```

There are precedence rules for Boolean operators. The `not` operator is highest, followed by `and`, then `or`.

Comparison operations all have the same precedence, and are evaluated before any Boolean operators are applied.

Like arithmetic expressions, the components of a Boolean expression are evaluated from left to right, accounting for precedence rules. Also like arithmetic operations, parentheses may be used for grouping, either to change the outcome or to improve the clarity for human readers.

How do we use these expressions to make decisions? We combine them with an `if` statement. We must now start to learn syntax, the correct way to write a statement. The syntax for an if block is

```
if condition_1:
     statement1
```

```
     statement2
elif condition_2:          #optional
     statement3
     statement4
else:                      #optional
     statement5
     statement6
```

The expression condition_1 must evaluate to `True` or `False`. If it is `True`, the next *block* of statements is executed. A block is a group of statements that is logically related. In Python, a block *must* be indented. The number of spaces must be exactly the same for each level of indentation. Four spaces is recommended, though not required (one space is the minimum); please do not use tabs. The colon after the `if`/`elif`/`else` statements is also required. A single statement may follow the colon on the same line, but frequently we place it on a separate line with an indentation, since that is usually easier to read and also makes it quicker to add new statements should we later need to do so.

If `condition_1` is `False` we skip to the next statement. If we want to carry out another test then we use `elif` (else if). If `condition_2` is `True` we execute the `elif` block. If it is `False` we skip that block. If we have code that we wish to execute only if both `condition_1` and `condition_2` are `False`, then we use an `else`. If we do not need this test, we go back to the outer level of code. We are not limited to a single `elif`, we may use as many as we need; but at most one `else` is permitted.

Loops

Now let us consider repetition. We need to repeat commands in many circumstances. We may need to apply an operation to every item in a list. We may need to process through the lines of a file. All this is called *looping*.

Python has two forms of loop, the `for` loop and the `while` loop. We will first consider the `for` loop. The syntax is

```
for item in iterable:
      statement1
      statement2
```

As for the `if` statement, a colon is required, followed by either a single statement or a code block.

An *iteratable* is a structure through which our variable can step, one by one, so we *iterate* through the loop. Very frequently in Python the iterable is a list. The *item* is called a loop variable and it is a variable that takes on the values contained in the iterator. Since it is a variable the programmer chooses the name.

Recall our groceries list from Episode 2. We can step through the list with

```
for item in groceries:
      print item
```

We can nest loops within loops.

```
for item in groceries:
    for letter in item:
        print letter
```

Notice how we used the outer loop variable as the iterable in the inner loop. We can do this because `item` will take on the elements of `groceries` one by one, and each element of `groceries` is a string, and an ordered sequence can be an iterable, so it works.

Often we want to iterate through a list of integers. Rather than typing them out, we can use a built-in function in Python called `range`. Range takes up to three arguments. If only one argument is present, it's the *non-inclusive* upper bound and the lower bound is zero.

```
range(10)
0,1,2,3,4,5,6,7,8,9
```

If two arguments are present, the first is the lower bound. It must be smaller than the upper bound.

```
range(1,10)
1,2,3,4,5,6,7,8,9
```

If three arguments are present, the third is the increment. If an increment is needed then both first and second arguments must be specified even if the lower bound is zero. If the increment is positive, the first argument must be less than the second argument; if it is negative, the first argument must be greater than the second. Range can handle only integers.

```
range(10,2,-2)
10,8,6,4
```

Example:

```
x_list=[0.]*20
for i in range(len(x_list)):
     x_list[i]=float(i)+float(i*i)
print x_list
```

Now let's look at while loops. A `for` loop executes a fixed number of iterations, whereas a while loop executes until some condition becomes `False`. The general form is

while condition:

statements

change condition

The condition must be changed or the loop will execute forever (an *infinite loop* situation). Occasionally this is done intentionally and we use the ability to break out of a loop, which we'll see later, to stop the execution. More commonly, an infinite loop is an error.

The interpreter evaluates the condition. If it is `True`, it executes all the statements in the following code block. It then re-evaluates the condition, using updated values. If it is `True` it continues; if it has become `False` it skips all the of code block and goes on with the rest of the program.

Study this example:

```
i=0
while i<100:
    print i
    i+=1
```

What happens when you change the condition to `i<=100`? What would happen if you started with `i=101`?

Early Termination

If we need to jump out of a loop before it would normally terminate, we can insert a break statement.

```
i=0
while i<20:
     i+=1
     if i==19:
          break
     print i
```

Or

```
for i in range(20):
     if i==19:
```

```
            break
```

If we wish to skip over the body of the loop from some point and return to the top, we use `continue`.

Example:

```
i=0
while i<20:
      i+=1
      if i==19:
          continue
      print i
```

This episode covers the most fundamental programming constructs, so be sure you understand it before you go on.

Further References

A good introduction to loops and conditionals is available at https://en.wikibooks.org/wiki/A_Beginner's_Python_Tutorial/Loops,_Conditionals