Lesson 5: Control Structures

Control structures are statements that allow a program to execute certain lines of code based on what happens in the code.

1 Conditionals

A conditional is a statement that can be evaluated as either *True* or *False*. Python already has a few things preprogrammed as *True* or *False*.
The following are *True*:

- non-zero numbers
- non-empty lists, tuples, and dictionaries
- non-empty strings
- the word *True*

The following are *False*:

- zero: 0, 0.0
- empty lists, tuples, and dictionaries: [ ], ( ), { }
- empty strings: ' ', " ", "' "'
- the word *False*

To check if an expression is *True* or *False*, Python uses comparisons:

`==`	equivalence (equal to)
`!=`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

We need to be careful. There is a difference between `=`, used for variable assignment, and `==`, used for checking equivalency. For example, we can write `x = 5`, which assigns the variable `x` to the integer `5`. But `x == 5` checks to see if `x` is equal to `5`, giving a result of `True` or `False`, depending on whether `x` had been previously set to `5`. Using `=` in place of `==` is a common mistake in Python coding.

2 Control Structures

`if`, `elif`, and `else`

One of the most often used control structures is the `if` statement. The basic syntax is

`if` *expression* :
    code to execute

If *expression* evaluates to `True`, then the indented code after the colon will be executed. If *expression* evaluates to `False`, then the indented code after the colon will *not* be executed. An example is shown in Figure 1. Keep in mind that each line we want executed by the

```
1 name = raw_input('Your first name: ')
2
3 if name[0] != 'D':
4     print 'Your name is not David'
5
```

Figure 1: A simple Python routine using the `if` statement.

`if` statement must be indented. When Python reaches a non–indented line, it will treat the line as outside the `if` statement and thus, as just a regular line of code.

Note that Canopy will take care of indenting the code for us, but a different text editor may not. This is one of the reasons why it is important to stick to one text editor. We can use the tab key to indent all the lines of code we want executed in the `if` statement, or we can use a series of spaces (usually four). If we switch text editors mid–code, the amount of space given to the tab may be different in the two editors. When Python tries to execute the program, the spacing will be inconsistent and Python may simply return an error.

We can extend the versatility of the `if` statement by using `elif` and `else`. The `elif` keyword is short for"else if". We can add these keywords like this:

`if` *expression one* :
    code to execute if expression one is true
`elif` *expression two* :
    code to execute if expression two is true
`else` :
    code to execute if neither expression one nor two is true

The `elif` keyword adds another expression to be evaluated if the first expression evaluates to `False`. Finally, the `else` keyword adds code that will be executed only if the expressions in the `if` and `elif` branches evaluate to `False`. We can add as many `elif` branches as we wish, but we can only have one `if` and one `else` branch.

Note that the *code to execute* can consist of multiple lines of code. Also note that we can nest `if` statements inside any of the `if`, `elif` and `else` branches. The code to be executed inside a nested statement needs to be indented further.

Exercise 1: Write a program that asks the user for their favorite color, and outputs a separate message depending on the choice. For example, if the color is red, the output might say `Wolfpack red!`. If the color is blue, the output might say `The sky is blue`. Include at least four color choices.

`and`, `or`, `not` and `in`

Python includes some boolean (logical) operators that can be included in control structures. These can help alleviate the need for nesting statements.

- `and` is used when we want to make sure two expressions are `True`
- `or` is used when we want to make sure at least one of two expressions is `True` but it doesn't matter which one
- `not` is used when we want the code to be executed when an expression is `False`
- `in` is used to test whether a variable is contained in a sequence

See Figure 2 for examples of these operators.

```
1  LastName = raw_input('What is your last name? ')
2  FirstName = raw_input('What is your first name? ')
3
4  if (LastName == 'Simpson') and FirstName:
5      print 'I see you are part of the Simpson family'
6      if FirstName[0] == 'M':
7          if ('r' in FirstName):
8              print 'You must be Marge'
9          else:
10             print 'You must be Maggie'
11     elif (FirstName[0] == 'H') or (FirstName[0] == 'B'):
12         if not (FirstName[0] == 'B'):
13             print 'Hello Homer'
14         else:
15             print 'Hello Bart'
16     else:
17         print 'Hi Lisa'
18 else:
19     print 'You\'re not part of the Simpson family'
20
```

Figure 2: Example code using boolean operators.

Exercise 2: Copy the code from figure 2 and run it. Make sure you understand what it is doing.

`while` and `for`

It is often useful to repeat a block of code several times until certain conditions are met. For this situation, we can use `while` or `for` statements. The `while` command looks like this:

`while` *condition* :
    code to execute while condition is True
code not part of `while` *statement*

The indented block of code repeats as long as the *condition* evaluates to `True`. For example:

```
a = 1
while a <= 5:
    print a**2
    a = a + 1
```

will produce the output

```
1
4
9
16
25
```

The `for` command takes the form

`for` *item* `in` *sequence* :
    code to execute

The indented block of code repeats for each *item* in the *sequence*. For example,

```
for a in range(1,6,1):
    print a**2
```

will produce the same output as above. Note that the `range(1,6,1)` command produces the array `[1,2,3,4,5]`.

The `while` and `for` statements work with other keywords, including:

- `pass`: this keyword actually does nothing, but can be used when an indented line is necessary
- `continue`: this keyword is used to move the program execution to the top of the while statement without finishing the portion of code after the continue keyword
- `break`: this keyword is used to "break out" of the statement and move the program execution to just after the statement

For example, consider the code

```
for i in range(1,20,1):
    if i < 5:
        continue
    elif i > 10:
        break
    print i
```

The command `range(1,20,1)` produces the array `1,2,...,19`. The `for` loops through the indented block of code once for each value of `i` in the array. The `continue` command causes the execution to return to the top of the `for` loop with the next `i` value. The `break` command ends the execution of the `for` loop.

The same result can be obtained with the following block of code:

```
for i in range(1,20,1):
    if i < 5:
        pass
    elif i > 10:
        pass
    else: print i
```

This illustrates the use of the `pass` keyword.

Exercise 3: Run the Python codes from this section. Make sure you understand the output.

Exercise 4: Write a program that asks for an integer from the user. The code should print out `1 and...` then `2 and...`, *etc* until it reaches the user's integer. For example, if the user's integer is 4, the program should produce:

1 and...
2 and...
3 and...
4 and...
done

Modify your code to break from the loop once the count reaches `10`.

Here is another example:

```
import numpy as n
v0 = 100.0 # initial velocity [m/s]
y0 = 1.0   # initial position [s]
g = 9.80   # acceleration [m/s^2]
t = n.linspace(0.0,100.0,1001) # array of times in 0.1 s increments
y = n.zeros(len(t))            # array of zeros, same length as t

# find positions y[i] at times t[i]
```

```
for i in range(len(t)):
    y[i] = y0 + v0*t[i] - 0.5*g*t[i]**2
```

This code creates an array of projectile heights `y` corresponding to the array `t` of times. The numpy command `y = n.zeros(len(t))` defines `y` as an array with the same number of elements as `t`. Initially, each array element in `y` is set to zero. The command `range(len(t))` is the same as `range(0,len(t),1)`. It gives an array `[0,1,2,...]` with the same number of elements as `t`.

Exercise 5: Execute this code. Add the `pylab` module and plot a graph of `y` versus `t`.

The code above carries out the same calculation as the code presented at the end of Section 2 of Lesson 3 (Modules, Arrays and Plots). The code from Lesson 3 is *much* more efficient, since it makes use of `numpy's` ability to carry out arithmetic operations directly on arrays. On the other hand, the code here uses an explicit `for` loop to carry out the calculations one array element at a time.

Homework: Write a program that asks the user for a positive integer x, and then computes all the prime numbers up to that integer. (You can assume $x > 2$ if you like.) The program should print out each prime number less than or equal to x, and then print one of the two statements: "Congratulations! The number you chose was prime!" or "Sorry, the number you chose was not prime." Your LaTeX document should include a *brief* (not detailed) explanation of the logic of your code, as well as the code itself.

There are many ways to solve this homework problem. It requires some careful thought. You will probably want to use some Python commands that have not been presented in any of the lessons. For example, you might need a command that will *append* an element to an array. You can find this command, and others, on the internet.