03_ControlFlow

November 13, 2019

1 Control Flow

We now know how to set variables of various types:

```
[ ]: a = 1
     b = 3.14
     c = 'hello'
     d = [a, b, c]
```

but this doesn't get us very far. One essential part of programming is **control flow** which is the ability to control how the program will proceed based on for example some conditions, or making parts of the program run multiple times.

1.1 `if` statements

The simplest form of control flow is the `if` statement, which executes a block of code only if a certain condition is true (and optionally executes code if it is *not* true. The basic syntax for an if-statement is the following:

```
if condition:
    # do something
elif condition:
    # do something else
else:
    # do yet something else
```

Notice that there is no statement to end the if statement, and the presence of a colon (`:`) after each control flow statement. Python relies on **indentation and colons** to determine whether it is in a specific block of code.

For example, in the following code:

```
[ ]: a = 3

     if a == 1:
         print("a is 1, changing to 2")
         a = 2
         if a==2:
             print()
     print("finished")
     a
```

The first print statement, and the `a = 2` statement only get executed if `a` is 1. On the other hand, `print "finished"` gets executed regardless, once Python exits the if statement.

Indentation is very important in Python, and the convention is to use four spaces (not tabs) for each level of indent.

Back to the if-statements, the conditions in the statements can be anything that returns a boolean value. For example, `a == 1`, `b != 4`, and `c <= 5` are valid conditions because they return either `True` or `False` depending on whether the statements are true or not.

Standard comparisons can be used (`==` for equal, `!=` for not equal, `<=` for less or equal, `>=` for greater or equal, `<` for less than, and `>` for greater than), as well as logical operators (`and`, `or`, `not`). Parentheses can be used to isolate different parts of conditions, to make clear in what order the comparisons should be executed, for example:

```
if (a == 1 and b <= 3) or c > 3:
    # do something
```

More generally, any function or expression that ultimately returns `True` or `False` can be used.

1.2 `for` loops

Another common structure that is important for controling the flow of execution are loops. Loops can be used to execute a block of code multiple times. The most common type of loop is the `for` loop. In its most basic form, it is straightforward:

```
for value in iterable:
    # do things
```

The `iterable` can be any Python object that can be iterated over. This includes lists or strings.

[ ]:
```
list = [2,12]
for a in list:
    print(a)
```

[ ]:
```
for letter in 'hello':
    print(letter)
```

A common type of for loop is one where the value should go between two integers with a specific set size. To do this, we can use the `range` function. If given a single value, it will allow you to iterate from 0 to the value minus 1:

[ ]:
```
for i in range(1,10):
    print(i)
```

[ ]:
```python
for i in range(3, 12):
    print(i)
```

[ ]:
```python
for i in range(2, 20, 2):  # the third entry specifies the "step size"
    print(i)
```

If you try iterating over a dictionary, it will iterate over the **keys** (not the values), in no specific order:

[ ]:
```python
d = {'a':['apple', 'grape'], 'b':2, 'c':3}
for rafael in d:
    if len(d[rafael])>1:
       print(d[rafael][0])
```

But you can easily get the value with:

[ ]:
```python
for key in d:
    print(key, d[key])
```

1.3 Building programs

These different control flow structures can be combined to form programs. For example, the following program will print out a different message depending on whether the current number in the loop is less, equal to, or greater than 10:

[ ]:
```python
for value in [2, 55, 4, 5, 12, 8, 9, 22]:
    if value > 10:
        print("Value is greater than 10 (" ,value ,")")
    elif value == 10:
        print("Value is exactly 10")
    else:
        print("Value is less than 10 (" + str(value) + ")")
```

1.4 Exercise 1

Write a program that will print out all the prime numbers (numbers divisible only by one and themselves) below 1000.

Hint: the % operator can be used to find the remainder of the division of an integer by another:

[ ]:
```python
if 20 % 6 != 0:
```

[ ]:
```python
for n in range(1,1000):
    prime = True
    for m in range(2,n):
        if n%m==0:
            prime=False
    if prime:
        print(n)
# enter your solution here
```

1.5 Exiting or continuing a loop

There are two useful statements that can be called in a loop - `break` and `continue`. When called, `break` will exit the loop it is currently in:

[ ]:
```
for i in range(10):
    print(i)
    if i == 3:
        break
```

The other is `continue`, which will ignore the rest of the loop and go straight to the next iteration:

[ ]:
```
for i in range(10):
    if i == 2 or i == 8:
        continue
    print(i)
```

1.6 Exercise 2

When checking if a value is prime, as soon as you have found that the value is divisble by a single value, the value is therefore not prime and there is no need to continue checking whether it is divisible by other values. Copy your solution from above and modify it to break out of the loop once this is the case.

[ ]:
```
for n in range(1,1000):
    prime = True
    for m in range(2,n):
        if n%m==0:
            prime=False
            break
    if prime:
        print(n)

# enter your solution here
```

1.7 `while` loops

Similarly to other programming languages, Python also provides a `while` loop which is similar to a `for` loop, but where the number of iterations is defined by a condition rather than an iterator:

```
while condition:
    # do something
```

For example, in the following example:

```
[ ]: a = 1
     while a < 10:
         print(a)
         a = a * 1.5
     print("Once the while loop has completed, a has the value", a)
```

the loop is executed until `a` is equal to or exceeds 10.

1.8 Exercise 3

Write a program (using a `while` loop) that will find the Fibonacci sequence up to (and excluding) 100000. The two first numbers are 0 and 1, and every subsequent number is the sum of the two previous ones, so the sequence starts `[0, 1, 1, 2, 3, 5, ...]`.

Optional: Store the sequence inside a Python list, and only print out the whole list to the screen once all the numbers are available. Then, check whether any of the numbers in the sequence are a square (e.g. `0*0`, `1*1`, `2*2`, `3*3`, `4*4`) and print out those that are.

```
[ ]: import math
     l = []
     #l2 = []
     # enter your solution here
     a=0
     b=1

     while b<100000:
         c = a+b
         l.append(a)
         a = b
         b = c

     #for x in l:
     #    d = math.sqrt(x)
     #    if d==int(d):
     #        l2.append(x)
     print(l)
     #print(l2)
```