**Python Lab 3**

**While loops**

Presenting our first *control structure*. Ordinarily the computer starts with the first line and then goes down from there. Control structures change the order that statements are executed or decide if a certain statement will be run. Here's the source for a program that uses the while control structure:

a = 0 *# FIRST, set the initial value of the variable a to 0(zero).*

**while** a < 10: *# While the value of the variable a is less than 10 do the following:*

 a = a + 1 *# Increase the value of the variable a by 1, as in: a = a + 1!*

 **print**(a) *# Print to screen what the present value of the variable a is.*

 *# REPEAT! until the value of the variable a is equal to 9!? See note.*

 *# NOTE:*

 *# The value of the variable a will increase by 1*

 *# with each repeat, or loop of the 'while statement BLOCK'.*

 *# e.g. a = 1 then a = 2 then a = 3 etc. until a = 9 then...*

 *# the code will finish adding 1 to a (now a = 10), printing the*

 *# result, and then exiting the 'while statement BLOCK'.*

 *# --*

 *# While a < 10: |*

 *# a = a + 1 |<--[ The while statement BLOCK ]*

 *# print (a) |*

 *# --*

And here is the extremely exciting output:

1

2

3

4

5

6

7

8

9

10

(And you thought it couldn't get any worse after turning your computer into a five-dollar calculator?)

So what does the program do? First it sees the line a = 0 and sets a to zero. Then it sees while a < 10: and so the computer checks to see if a < 10. The first time the computer sees this statement, a is zero, so it is less than 10. In other words, as long as a is less than ten, the computer will run the tabbed in statements. This eventually makesa equal to ten (by adding one to a again and again) and the while a < 10 is not true any longer. Reaching that point, the program will stop running the indented lines.

Always remember to put a colon "**:**" at the end of the while statement line!

Here is another example of the use of while:

a = 1

s = 0

**print**('Enter Numbers to add to the sum.')

**print**('Enter 0 to quit.')

**while** a != 0:

 **print**('Current Sum:', s)

 a = float(input('Number? '))

 s = s + a

**print**('Total Sum =', s)

Enter Numbers to add to the sum.

Enter 0 to quit.

Current Sum: 0

Number? **200**

Current Sum: 200.0

Number? **-15.25**

Current Sum: 184.75

Number? **-151.85**

Current Sum: 32.9

Number? **10.00**

Current Sum: 42.9

Number? **0**

Total Sum = 42.9

Notice how print 'Total Sum =', s is only run at the end. The while statement only affects the lines that are indented with whitespace. The != means does not equal sowhile a != 0: means as long as a is not zero run the tabbed statements that follow.

Note that a is a floating point number, and not all floating point numbers can be accurately represented, so using != on them can sometimes not work. Try typing in 1.1 in interactive mode.

**What will this program do??**

**print("Hello World")**

**#stops console from exiting**

**end\_prog = ""**

**while end\_prog != "q":**

 **end\_prog = input("type q to quit")**

 **print("Hello World")**

**Infinite loops or Never Ending Loop**

Now that we have while loops, it is possible to have programs that run forever. An easy way to do this is to write a program like this:

**while** 1 == 1:

 **print**("Help, I'm stuck in a loop.")

The "==" operator is used to test equality of the expressions on the two sides of the operator, just as "<" was used for "less than" before (you will get a complete list of all comparison operators in the next lab).

This program will output Help, I'm stuck in a loop. until the heat death of the universe or you stop it, because 1 will forever be equal to 1. The way to stop it is to hit the Control (or *Ctrl*) button and *C* (the letter) at the same time. This will kill the program. (Note: sometimes you will have to hit enter after the Control-C.) On some systems, nothing will stop it, short of killing the process--so avoid!

**Examples**

**Fibonacci sequence**

**Fibonacci-method1.py**

*# This program calculates the Fibonacci sequence*

a = 0

b = 1

count = 0

max\_count = 20

**while** count < max\_count:

 count = count + 1

 **print**(a, end=" ") *# Notice the magic end=" " in the print function arguments*

 *# that keeps it from creating a new line.*

 old\_a = a *# we need to keep track of a since we change it.*

 a = b

 b = old\_a + b

**print**() *# gets a new (empty) line.*

Output:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

Note that the output is on a single line because of the extra argument end=" " in the print arguments.

**Fibonacci-method2.py**

*# Simplified and faster method to calculate the Fibonacci sequence*

a = 0

b = 1

count = 0

max\_count = 10

**while** count < max\_count:

 count = count + 1

 **print**(a, b, end=" ") *# Notice the magic end=" "*

 a = a + b

 b = a + b

**print**() *# gets a new (empty) line.*

Output:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181

**Enter password**

**Password.py**

*# Waits until a password has been entered. Use Control-C to break out without*

*# the password*

*#Note that this must not be the password so that the*

*# while loop runs at least once.*

password = str()

*# note that != means not equal*

**while** password != "unicorn":

 password = input("Password: ")

**print**("Welcome in")

Sample run:

Password: **auo**

Password: **y22**

Password: **password**

Password: **open sesame**

Password: **unicorn**

Welcome in

**Exercises**

Using a **While loop**, write a program that asks the user for a Login Name and password. Then when they type "lock", they need to type in their name and password to unlock the program.

**For Loops**

And here is the new typing exercise for this lab:

onetoten = range(1, 11)

**for** count **in** onetoten:

 **print**(count)

and the ever-present output:

1

2

3

4

5

6

7

8

9

10

The output looks awfully familiar but the program code looks different. The first line uses the range function. The range function uses two arguments like this range(start, finish). start is the first number that is produced. finish is one larger than the last number. Note that this program could have been done in a shorter way:

**for** count **in** range(1, 11):

 **print**(count)

The range function returns an iterable. This can be converted into a list with the list function. Here are some examples to show what happens with the range command:

>>> **range(1, 10)**

range(1, 10)

>>> **list(range(1, 10))**

[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> **list(range(-32, -20))**

[-32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21]

>>> **list(range(5,21))**

[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]

>>> **list(range(5))**

[0, 1, 2, 3, 4]

>>> **list(range(21, 5))**

[]

The next line for count in onetoten: uses the for control structure. A for control structure looks like for variable in list:. list is gone through starting with the first element of the list and going to the last. As for goes through each element in a list it puts each into variable. That allows variable to be used in each successive time the for loop is run through. Here is another example (you don't have to type this) to demonstrate:

demolist = ['life', 42, 'the universe', 6, 'and', 7, 'everything']

**for** item **in** demolist:

 **print**("The current item is:",item)

The output is:

The current item is: life

The current item is: 42

The current item is: the universe

The current item is: 6

The current item is: and

The current item is: 7

The current item is: everything

Notice how the for loop goes through and sets item to each element in the list. So, what is for good for? The first use is to go through all the elements of a list and do something with each of them. Here's a quick way to add up all the elements:

list = [2, 4, 6, 8]

sum = 0

**for** num **in** list:

 sum = sum + num

**print**("The sum is:", sum)

with the output simply being:

The sum is: 20

Or you could write a program to find out if there are any duplicates in a list like this program does:

list = [4, 5, 7, 8, 9, 1, 0, 7, 10]

list.sort()

prev = None

**for** item **in** list:

 **if** prev == item:

 **print**("Duplicate of", prev, "found")

 prev = item

and for good measure:

Duplicate of 7 found

Okay, so how does it work? Here is a special debugging version to help you understand (you don't need to type this in):

l = [4, 5, 7, 8, 9, 1, 0, 7, 10]

**print**("l = [4, 5, 7, 8, 9, 1, 0, 7, 10]", "**\t\t**l:", l)

l.sort()

**print**("l.sort()", "**\t\t**l:", l)

prev = l[0]

**print**("prev = l[0]", "**\t\t**prev:", prev)

**del** l[0]

**print**("del l[0]", "**\t\t**l:", l)

**for** item **in** l:

 **if** prev == item:

 **print**("Duplicate of", prev, "found")

 **print**("if prev == item:", "**\t\t**prev:", prev, "**\t**item:", item)

 prev = item

 **print**("prev = item", "**\t\t**prev:", prev, "**\t**item:", item)

with the output being:

l = [4, 5, 7, 8, 9, 1, 0, 7, 10] l: [4, 5, 7, 8, 9, 1, 0, 7, 10]

l.sort() l: [0, 1, 4, 5, 7, 7, 8, 9, 10]

prev = l[0] prev: 0

del l[0] l: [1, 4, 5, 7, 7, 8, 9, 10]

if prev == item: prev: 0 item: 1

prev = item prev: 1 item: 1

if prev == item: prev: 1 item: 4

prev = item prev: 4 item: 4

if prev == item: prev: 4 item: 5

prev = item prev: 5 item: 5

if prev == item: prev: 5 item: 7

prev = item prev: 7 item: 7

Duplicate of 7 found

if prev == item: prev: 7 item: 7

prev = item prev: 7 item: 7

if prev == item: prev: 7 item: 8

prev = item prev: 8 item: 8

if prev == item: prev: 8 item: 9

prev = item prev: 9 item: 9

if prev == item: prev: 9 item: 10

prev = item prev: 10 item: 10

The reason I put so many print statements in the code was so that you can see what is happening in each line. (By the way, if you can't figure out why a program is not working, try putting in lots of print statements in places where you want to know what is happening.) First the program starts with a boring old list. Next the program sorts the list. This is so that any duplicates get put next to each other. The program then initializes a prev(ious) variable. Next the first element of the list is deleted so that the first item is not incorrectly thought to be a duplicate. Next a for loop is gone into. Each item of the list is checked to see if it is the same as the previous. If it is a duplicate was found. The value of prev is then changed so that the next time the for loop is run through prev is the previous item to the current. Sure enough, the 7 is found to be a duplicate. (Notice how \t is used to print a tab.)

The other way to use for loops is to do something a certain number of times. Here is some code to print out the first 9 numbers of the Fibonacci series:

a = 1

b = 1

**for** c **in** range(1, 10):

 **print**(a, end=" ")

 n = a + b

 a = b

 b = n

with the surprising output:

1 1 2 3 5 8 13 21 34

Everything that can be done with for loops can also be done with while loops but for loops give an easy way to go through all the elements in a list or to do something a certain number of times.