Lab 2: Basic Flow Control

CSE/IT 107

NMT Computer Science

"When you come to a fork in the road, take it."

— Attributed to Yogi Berra

"Simplicity is the ultimate sophistication."

— Leonardo Da Vinci

"How do we convince people that in programming simplicity and clarity – in short: what mathematicians call elegance – are not a dispensable luxury, but a crucial matter that decides between success and failure?"

— Edsger Dijkstra

---

1 Introduction

The purpose of this lab is to introduce you to the fundamentals of what programmers call flow control. In the previous lab, we showed you how to do basic calculations in Python. For example, we had you convert temperature from Celsius to Fahrenheit and Kelvin.

What if the user of your conversion program wanted to have only one conversion and you did not know which? We have to be able to give the user a choice. In the previous lab, you learned about the `input()` function that let you "ask" the user of your program a question. In this lab, you will learn how to use `if`, `else`, and `elif` to have the program choose one action out of multiple actions; for example, whether to convert to Kelvin or to Fahrenheit.

Sometimes, you also want to be able to repeat a calculation for different values. For example, you want to calculate the square root of all numbers between 1 and 100. To do this, you do not have to actually repeat writing the calculation in your code, there is the `while` statement to help you repeat code.

1.1 New Code Coloring in PDFs

In the new code style, all variable names will be black, all keywords will be blue, all strings will be maroon (red), while comments are green. Any code that is run in the Python interactive interpreter is on grey background.

2 Boolean Logic

A common activity when programming is determining if some value is true or false. For example, checking if a variable is less than five or if the user entered the correct password. Any statement that can be resolved into a true or a false value is called a boolean statement, the value it resolves into (true or false) is called a boolean value.

```
>>> x = 5
>>> print(x < 3)
False
>>> print(x < 6)
True
```

In the above example, the boolean values are `True` and `False`. The boolean statements are `x < 3` and `x < 6`.

In addition to <, we can also test for other inequalities.

```
>>> x = 3; y = 6
>>> print(x < y)
True
>>> print(x > y)
False
>>> print(x <= y)
True
>>> print(x >= y)
False
```

Note that <= means "less than or equal to" and >= means "greater than or equal to".

Finally, we can test if two values are equal (==) or not equal (`!=`).

```
>>> x = 3; y = 3; z = 4
>>> print(x == y)
True
>>> print(x == z)
False
>>> print(y != 5)
True
>>> print(y != x)
False
```

It is important to remember that we use = to assign a value to a variable and == to test if two values are equal.

We can also combine boolean statements using `and` and `or` as such:

```
>>> x = 3; y = 5; z = 8
>>> print(x < y and y < z)
True
>>> print(x > y and y < z)
False
>>> print(x > y or y < z)
True
>>> print(True and False)
False
```

```
>>> print(True or False)
True
```

If you combine two boolean statements that are true using `and`, the result will be true. In all other cases the result is false. Since `x < y` is true and `y < z` is true, we have that `x < y and y < z` is true. See Table 1 for a "truth table" showing what combinations are true or false – it should make sense though.

In addition to this, there is the `not` operator to negate a boolean statement. You can also put a boolean statement in parentheses to do more complicated combinations:

```
>>> x = 3; y = 5; z = 8
>>> print(not True)
False
>>> print(not (x > y and y < z))
True
```

A	B	A and B	A or B	not A
True	True	True	True	False
True	False	False	True	False
False	True	False	True	True
False	False	False	False	True

Table 1: Truth table

2.1 Summary

Comparison operator	What it tests
`a < b`	is *a* less than *b*
`a > b`	is *a* greater than *b*
`a <= b`	is *a* less than or equal to *b*
`a >= b`	is *a* greater than or equal to *b*
`a == b`	is *a* equal to *b*
`a != b`	is *a* not equal to *b*

Table 2: Comparison operators

Boolean combination operator	What it does
`a and b`	true if both *a* and *b* are true
`a or b`	true if either *a* or *b* are true or both
`not a`	true if *a* is false

Table 3: Combination operators

3 Conditional Statements

The primary use for boolean values is to determine which branch in your code to follow. This is accomplished using `if` and `else`, as shown in the program below. `elif` will be introduced later in this lab. All three – `if`, `elif`, and `else` – are generally called *conditional statements*.

```
x = 1
y = float(input("Please input a number: "))

if x == y:
    print("x and y are equal.")
else:
    print("x and y are not equal.")
print("When do I print?")
```

Try running the above program, putting in different numbers for `y`. If the number input is 1, then the first `print` statement will output. If not, then the second one will. The third one will output regardless.

The way this works is very simple: either the first `print` statement runs or the second `print` statement runs, but never both. Which one runs is determined by Python: if the boolean statement (called *condition*) following the `if` evaluates to `True`, then Python will run the indented code following the `if` and then skip until after the indented code of the `else`.

However, if the *condition* evaluates to `False`, then the indented code following the `else` is run and Python skips the indented code between `if` and `else`.

It is important to note that the code that follows `if` or `else` **must** be indented.

See what happens when you run the code below compared to the other piece of code above:

```
x = 1
y = float(input("Please input a number: "))

if x == y:
    print("x and y are equal.")
else:
    print("x and y are not equal.")
    print("When do I print?")
```

There are many uses for conditional statements, such as to ensure that a given variable is not negative:

```
x = float(input("Please input a number: "))

if x < 0:
    x = 0 # sets x equal to 0 if x was less than 0

print("x = " + str(x))
```

You can perform other operations as part of a boolean statement, such as this convenient way to check if a number is even:

```
x = 5

if x % 2 == 0:
```

```
    print("x is even.")
else:
    print("x is odd.")
```

Remember that % is the modulus operator: it gives you the remainder of the division.

When using `if` and `else`, you will generally be dealing with user input. This is done using the function `input`, which you can see used in the above examples. When you use `input` it will display whatever string you pass to it, then pause while it waits for the user to type something and then hit enter. It will give whatever was entered as a string back to the variable that it is assigned to. We will be learning more about strings in future labs, but for now just know that strings are basically groups of letters, like what you pass to a `print` statement, and are declared by surrounding something in quotes.

The main thing to know about strings for now is that they cannot be used as numbers. This is why we use the `float` function to convert the string the user gives us into a number.

```
>>> 5.5 == "5.5"
False
>>> 5.5 == float("5.5")
True
```

It is important to understand the order that things happen in a statement like

`x = float(input("Please input a number: "))`

Though both `x =` and `float` appear first in the line, the first statement to execute is `input`. This is because `input` is inside of `float`'s parentheses and is therefore being passed as a parameter to `float`. Therefore, `float` cannot run until `input` is finished and has returned a value to be used by `float`. Similarly, `x =` will not happen until `float` has finished converting the value into a number.

If you are comparing strings, then you do not need to go through the extra step of converting the user's input into a number:

```
password = "hunter2"

user_pass = input("Please input the password: ")

if password == user_pass:
    print("Password is correct. Welcome!")
else:
    print("Invalid password.")
```

In some cases, it could be that there are multiple passwords. Try running the following code:

```
password = "hunter2"
also_password = "hunter3"
another_password = "hunter4"
user_pass = input("Please input the password: ")

if password == user_pass:
    print("Welcome, administrator.")
elif user_pass == also_password:
    print("Welcome, administrator.")
elif user_pass == another_password:
    print("Welcome, manager.")
```

```
else:
    print("Wrong password.")
```

This introduced you to the `elif` statement: When the condition following `if` turns out to be false, Python will then check the first `elif` statement. If that condition turns out to be true, it will run the code following that `elif` statement or move on to the next `elif`. Only if none of the conditions turned out to be true, the code following `else` will be run.

If you remember the boolean logic section, this problem could have been solved more efficiently: two passwords are resulting in the same code. You could rewrite the code like this:

```
user_pass = input("Please input the password: ")

if user_pass == "hunter2" or user_pass == "hunter3":
    print("Welcome, administrator.")
elif user_pass == "hunter4":
    print("Welcome, manager.")
else:
    print("Wrong password.")
```

You can also nest the statements you just learned about. Try running the following code, trying multiple values:

```
x = float(input("Enter a value for x: "))
y = float(input("Enter a value for y: "))

if x > 0:
    if y > 0:
        print("Both are greater than 0.")
    else:
        print("x is greater than 0, but y is smaller or equal to 0")
else:
    print("x is smaller or equal to 0.")
```

3.1 Summary

- Conditional statements look like this:

```
if condition:
    # some code to run
elif othercondition:
    # some other code to run
else:
    # alternative code if no condition was met
```

- The `elif` and `else` sections are both optional
- `elif` statements can be repeated as many times as you want.
- The conditions must be boolean statements.
- The code inside `if`, `elif`, and `else` statements must be indented. Python will either show an error or behave very weirdly if you do not indent the code.

- You can nest conditional statements.

3.2 Exercises

conversions.py Use your `conversions.py` from last time and add a prompt asking the user whether to convert to Kelvin or Fahrenheit. It should look like this when it is run:

```
Please input the temperature in Celsius: 10
Please choose Kelvin (K) or Fahrenheit (F): F
You chose Fahrenheit.
Fahrenheit temperature: 50.0
```

```
Please input the temperature in Celsius: 10
Please choose Kelvin (K) or Fahrenheit (F): K
You chose Kelvin.
Kelvin temperature: 283.15
```

```
Please input the temperature in Celsius: 10
Please choose Kelvin (K) or Fahrenheit (F): E
You entered a letter I do not recognize.
```

calculator.py Write a small calculator that can compute arcsin, arccos, arctan and square root of a number. Use `math.sqrt()`, `math.asin()`, `math.acos()`, and `math.atan()`. Remember to import `math`.

Make sure to check for each function that the input is valid. For example, for square root the input cannot be negative. For arcsin, the input must be between -1 and 1 inclusive. Try to figure out what the input must be for arccos and arctan yourself!

```
Enter a number to use: 16
Which operation? sqrt (s), arcsin (a), arccos (c), arctan (t): s
The square root of 16 is 4.0.
```

```
Enter a number to use: 16
Which operation? sqrt (s), arcsin (a), arccos (c), arctan (t): a
16 is not in the domain of arcsin!
```

4 Formatting Strings

Previously, when we wanted to print out both a number and a string, we had to resort to this:

```
>>> x = 5
>>> print("x is equal to " + str(x))
x is equal to 5
```

However, there is an easier way to accomplish the same thing. By using the `.format()` command, as shown below, we can have far more options for how we format our output.

```
>>> x = 5
>>> print("x is equal to {}".format(x))
x is equal to 5
```

Rather than leaving a gap in our string and then using `+` to add on our variable, we instead include `{}` where we wish to place our variable and add on `.format(x)` to the end of the string. This replaces `{}` with the value of `x`.

If we include multiple instances of `{}` in our string, we can then pass multiple variables to `.format()`. It will place them in the string in the order provided.

```
>>> x = 5
>>> y = 6
>>> print("x is equal to {} and y is equal to {}.".format(x, y))
x is equal to 5 and y is equal to 6.
```

We can also use `.format()` to control our output. For example, we can restrict how many decimal places a floating point number is printed with. To do this, we add `:.2f` inside of the `{}`. The `.2f` specifies that we want 2 digits to follow the decimal point.

```
>>> import math
>>> print(math.pi)
3.141592653589793
>>> print("{:.2f}".format(math.pi))
3.14
```

For more format options, see

`https://docs.python.org/3.1/library/string.html#format-string-syntax`

4.1 Summary

- Syntax:

```
print("string containing {}".format(variable))
```

  This will replace the `{}` with the value of `variable`.

- You can include multiple `{}` in a string and pass multiple values to `format`.
- You can specify advanced formatting options, such as number of digits after the decimal point.

5 Loops

5.1 While Loops

The syntax of a `while` loop is very similar to that of an `if` statement, but instead of only running the indented block of code once, the `while` loop will continue running it until the given boolean statement is no longer true.

```
1 x = 10
2
3 while x > 0:
4     print(x)
5     x = x - 1
```

The above program will print out the numbers 10 to 1. Try stepping through this program on paper, writing out the value of `x` at each time through the loop. Then repeat for this modified version of the program:

```
1 x = 10
2
3 while x > 0:
4     x = x - 1
5     print(x)
```

This version of the program will print out the numbers 9 to 0. This might seem a bit strange, since the condition of the loop says it will stop when `x` is no longer larger than 0. And yet, it prints out the value 0 before the loop ends. This is because the loop condition is only checked whenever the end of the indented section is reached. If the condition is `True`, then the indented section will be executed again. If the condition is `False`, then the loop will end.

If the condition starts out `False`, then the loop will never execute. The following program will not print anything:

```
1 x = 0
2
3 while x > 0:
4     x = x - 1
5     print(x)
```

`if` and `else` can be combined with `while`, as shown below:

```
1 x = 10
2
3 while x > 0:
4     if x % 2 == 0:
5         print("{} is even.".format(x))
6     else:
7         print("{} is odd.".format(x))
8     x = x - 1
```

Of course, they can be nested the other way around, too, with a `while` inside conditional statements.

There can also be infinite while loops. Try the following:

```
while True:
    print("Printing forever")
```

Press *Ctrl+C* to stop the execution of this.

5.2 For Loops

Instead of using `while` loops, you can also use the `for` iterator (often also called `for` loop). The `for` loop allows you to "iterate" over a given list of things, for example a list of characters (a string):

```
for c in "abc":
    print("Hi, {}!".format(c))
```

Here, `c` is a variable you can use inside the code block of the for loop.

The example will print:

```
Hi, a!
Hi, b!
Hi, c!
```

A for loop can be over numbers as well, but this requires us to use the `range()` function. You can give `range()` a starting number and an end number:

```
for num in range(0, 10):
    print(num)
```

This will print the numbers from *zero* to *nine*. Notice that 10 is not included!

You can also give `range()` a starting number, an end number, and an increment. The increment can be positive or negative. The numbers do not have to be actual numbers, you can give variables to it, too:

```
start = 10
end = 0
increment = -2
for number in range(start, end, increment):
    print(number)
```

This will print:

```
10
8
6
4
2
```

Notice how zero is not included. The end number is never included when you use `range()`.

You can also just have the `for` loop "iterate" over a list of things:

```
for number in [0, -2, 20, 24]:
    print(number)
```

5.3 Nesting

You can nest for loops and while loops and conditional statements in any way you like. The following is just an example:

```
parity = input("Even or odd? ")

# prints even or odd numbers between 0 and 10, depending on user input
if parity == "odd":
    for n in range(1, 11, 2):
        print(n)
elif parity == "even":
    n = 0
    while n <= 10:
        print(n)
        n = n + 2
else:
    print("You did not enter even or odd.")
```

5.4 Summary

- Syntax:

```
while condition:
    # code to be repeated
```

  This will repeat the indented code following the `while` until the condition is not true anymore. It checks the condition first, then runs the indented code, then checks the condition again, etc. Thus, if the condition is wrong in the first place, it will never run.

- Syntax:

```
for variable in list:
    # code doing something with variable
```

  The list can be a string or a `range()` function or an actual list delimited by brackets `[]`. You will soon learn that there is also a list data type in Python that you can use here and other nice things, but we do not cover that in this lab.

- There can be infinite while loops.

- You can nest conditional statements and loops any way you want in any combination.

5.5 Exercises

fizzbuzz.py Have the user enter a positive integer number. Then, print the numbers from 1 to that number each on a line. When the printed number is divisible by 3, print "Fizz", and when the number is divisible by 5, print "Buzz", and when it is divisible by both, print "FizzBuzz".

You must use `.format()` and a `while` loop.

Should look like this when run:

```
Enter a number: 16
1
2
3 Fizz
4
5 Buzz
6 Fizz
7
8
9 Fizz
10 Buzz
11
12 Fizz
13
14
15 FizzBuzz
16
```

```
Enter a number: -1
Not a positive number!
```

fizzbuzz_for.py Write the same fizzbuzz program using a `for` loop and `range()` this time. Using `.format()` is still required.

primes.py (Practice of Computing Using Python #2.15) Write a program that checks if a number N is prime. You have to ask the user for the number. Remember that a prime number is a number that is divisible only by 1 and itself.

A simple approach checks all numbers from 2 up to N.

Try to improve on the simple approach, though: do we really need to check all those numbers? At which point do you know that you can stop? Remember to import `math`.

dna.py DNA is generally encoded with four letters: A, T, G, and C. For example, a string of DNA would be "ATTGCAT".

You can find the "complement" of DNA for some biological reason (it is double stranded, but that does not matter to us). The complement of A is T and vice versa; the complement of G is C and vice versa.

Write a program that takes in a strand of DNA from the user using `input` and finds its complement. Assume the user always enters valid DNA made up of the four letters.

```
Enter DNA: ATTGCAT
Complement is: TAACGTA
```

6 Turtle

Some of the exercises for this lab will use Turtle, a simple graphics library. It can be accessed by using `import turtle` in Python. From there you have access to a group of functions for controlling the "turtle", a simple arrow that moves around at your command, drawing a line where it goes. The primary commands to control the turtle are shown in Table 4.

Operator	What it does
`turtle.forward(x)`	move the turtle forward `x` pixels
`turtle.backward(x)`	move the turtle backward `x` pixels
`turtle.left(x)`	turn the turtle left `x` degrees
`turtle.right(x)`	turn the turtle right `x` degrees

Table 4: Turtle commands

It should be noted that giving a negative value to these commands is allowed, so `turtle.left(-30)` is equivalent to `turtle.right(30)`.

Combining these commands will let you draw potentially complex shapes. For example, the following program will draw a hexagon.

```
1  import turtle
2
3  turtle.forward(100)
4  turtle.left(60)
5  turtle.forward(100)
6  turtle.left(60)
7  turtle.forward(100)
8  turtle.left(60)
9  turtle.forward(100)
10 turtle.left(60)
11 turtle.forward(100)
12 turtle.left(60)
13 turtle.forward(100)
14 turtle.left(60)
```

However, this code is a bit longer than it needs to be. Let's clean it up a bit using `while`.

```
1  import turtle
2
3  counter = 0
4
5  while counter < 6:
6      turtle.forward(100)
7      turtle.left(angle)
8      counter = counter + 1
```

6.1 Exercises

star.py Write a program that draws a shape like the picture below.

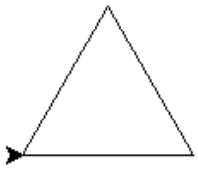
Hint: Try this on paper first. (It does not matter if the turtle is still displayed when it is drawn.)

polygons.py Write a program that takes in a number using `input` and then draws a regular polygon with that many sides. A regular polygon is one where each side is the same length and each corner is the same angle. For example, the sample code given in the section draws a regular hexagon.

Input:

```
1 How many sides? 3
```

Output:



navigate.py Write a program that takes directions from the command line to draw a line. Let the user input "left", "right", "forward", or "stop". Left and right turn the turtle left or right however many degrees are entered, forward moves the turtle forward (however far you wish), and stop ends the program. Please check the degrees for errors: they must be between 0 and 360 degrees! (Yes, Turtle could handle negative degrees, but we would like you to check.)

Input:

```
1 Please enter a direction: forward
2 Please enter a direction: left
3 How many degrees? 45
4 Please enter a direction: forward
5 Please enter a direction: left
6 How many degrees? -1
7 Invalid number, not moving.
8 Please enter a direction: left
9 How many degrees? 45
```

```
Please enter a direction: forward
Please enter a direction: forward
Please enter a direction: left
How many degrees? 45
Please enter a direction: left
How many degrees? 45
Please enter a direction: forward
Please enter a direction: right
How many degrees? 45
Please enter a direction: forward
Please enter a direction: stop
```

Output:

7 Submitting

Files to submit:

- conversions.py (see Section 3.2)
- calculator.py (see Section 3.2)
- fizzbuzz.py (see Section 5.5)
- fizzbuzz_for.py (see Section 5.5)
- primes.py (see Section 5.5)
- dna.py (see Section 5.5)
- polygons.py (see Section 6.1)
- navigate.py (see Section 6.1)
- star.py (see Section 6.1)

You may submit your code as either a tarball (instructions below) or as a .zip file. Either one should contain all files used in the exercises for this lab. The submitted file should be named either `cse107_firstname_lastname_lab2.zip` or `cse107_firstname_lastname_lab2.tar.gz` depending on which method you used.

For Windows, use a tool you like to create a `.zip` file. The TCC computers should have `7z` installed. For Linux, look at lab 1 for instructions on how to create a tarball or use the "Archive Manager" graphical tool.

Upload your tarball or .zip file to Canvas.