

Lab 2: Basic Flow Control

CSE/IT 107

NMT Department of Computer Science and Engineering

“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

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.

Contents

Introduction	i
1 Boolean Logic	1
1.1 Summary	2
2 On Types	3
2.1 Summary	4
3 Conditional Statements	5
3.1 Summary	8
3.2 Exercises	8
4 Formatting Strings	10
4.1 Summary	10
5 Loops	12
5.1 While Loops	12
5.2 Nesting	13
5.3 Summary	13
5.4 Exercises	14
6 Turtle	16
6.1 Exercises	17
7 Submitting	19

1 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.

```
1 >>> x = 5
2 >>> print(x < 3)
3 False
4 >>> print(x < 6)
5 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.

```
1 >>> x = 3; y = 6
2 >>> print(x < y)
3 True
4 >>> print(x > y)
5 False
6 >>> print(x <= y)
7 True
8 >>> print(x >= y)
9 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 (`!=`).

```
1 >>> x = 3; y = 3; z = 4
2 >>> print(x == y)
3 True
4 >>> print(x == z)
5 False
6 >>> print(y != 5)
7 True
8 >>> print(y != x)
9 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:

```
1 >>> x = 3; y = 5; z = 8
2 >>> print(x < y and y < z)
```

```

3  True
4  >>> print(x > y and y < z)
5  False
6  >>> print(x > y or y < z)
7  True
8  >>> print(True and False)
9  False
10 >>> print(True or False)
11 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:

```

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

```

A	B	A <code>and</code> B	A <code>or</code> B	<code>not</code> 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

1.1 Summary

Comparison operator	What it tests
<code>a < b</code>	is <i>a</i> less than <i>b</i>
<code>a > b</code>	is <i>a</i> greater than <i>b</i>
<code>a <= b</code>	is <i>a</i> less than or equal to <i>b</i>
<code>a >= b</code>	is <i>a</i> greater than or equal to <i>b</i>
<code>a == b</code>	is <i>a</i> equal to <i>b</i>
<code>a != b</code>	is <i>a</i> not equal to <i>b</i>

Table 2: Comparison operators

Boolean combination operator	What it does
<code>a and b</code>	true if both <i>a</i> and <i>b</i> are true
<code>a or b</code>	true if either <i>a</i> or <i>b</i> are true or both
<code>not a</code>	true if <i>a</i> is false

Table 3: Combination operators

2 On Types

In Python, every value is associated with what we call a type. We have already seen a few types in action: integers, floating-point numbers, strings, and boolean values. The type of a value restricts the set of things that it can represent.

An integer can be any whole number, for example 5. A float (floating-point number) is a number with decimal places, for example 3.14 or 5.0. A string is a sequence of characters (letters, numbers, ...) enclosed by either double or single quotes, for example "I'm a string!". A boolean, as we just learned, can have two values: True or False.

Most programming languages have types for a good reason: for one, operations (such as +, -, ...) have different effects on different types. For example, an integer * an integer results in an integer (the multiplication of the two *operands*), but an integer * a string results in the string repeated. However, a string * a string results in an error:

```

1 >>> 5*3
2 15
3 >>> 5*'hi'
4 'hihihihihi'
5 >>> 'hi'*'hi'
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   TypeError: can't multiply sequence by non-int of type 'str'

```

In addition to that, some operations do not work on some types, which helps ensure correctness of your program. Also, depending on the type of the operands, Python has different implementations of what goes on under the hood, which can then be optimized for each type separately.

Of course, there are ways to convert between types and there are ways to find out what type the value of a specific variable is. To do so, use the function `type()`.

```

1 >>> type(5)
2 <class 'int'>
3 >>> x = 10
4 >>> type(x)
5 <class 'int'>
6 >>> x = 'Allons-y!'
7 >>> type(x)
8 <class 'str'>
9 >>> type(True)
10 <class 'bool'>
11 >>> x = "5.5" # string containing 5.5

```

```

12 >>> y = float("5.5") # convert string with 5.5 to a float
13 >>> y
14 5.5
15 >>> z = int("5") # integer containing 5
16 >>> z
17 5
18 >>> q = str(5) # string containing 5
19 >>> q
20 '5'
21 >>> # Note that an impossible conversion will throw an error
22 >>> int("55a")
23 Traceback (most recent call last):
24   File "<stdin>", line 1, in <module>
25 ValueError: invalid literal for int() with base 10: '55a'

```

We call the process of converting a value from one type to another type *casting*. Note that only values have types, and variables can hold different values. Applying `type()` to a variable will give you the type of the value that it holds at the time. In the previous section, we learned about comparison operators such as `!=`, `<`, `>`, etc. What was not mentioned is that these comparison operators also compare the type in some cases, but not in others:

```

1 >>> 5.5 == '5.5' # comparing a string to a float does not work
2 False
3 >>> x = 5.5
4 >>> y = 5.5
5 >>> x == y
6 True
7 >>> 5.5 == float('5.5')
8 True
9 >>> 5.0 == 5 # comparing a float and an int works
10 True

```

The usual rule for what works and what does not work with comparisons is to listen to your intuition: it makes sense to compare different types of numbers; while it does not make sense to compare strings with numbers.

2.1 Summary

- A type is a set of values that it can represent.
- Casting is the process of converting a value from one type to another.
- To find out what type a value / variable is, use the `type()` function.
- Types we know about at this point:
 - `str` – string
 - `float` – floating-point number
 - `int` – integer number
 - `bool` – boolean value

Not coincidentally, these type names are also the names of the casting functions.

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*.

```
1 x = 1
2 y = float(input("Please input a number: "))
3
4 if x == y:
5     print("x and y are equal.")
6 else:
7     print("x and y are not equal.")
8 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:

```
1 x = 1
2 y = float(input("Please input a number: "))
3
4 if x == y:
5     print("x and y are equal.")
6 else:
7     print("x and y are not equal.")
8     print("When do I print?")
```

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

```
1 x = float(input("Please input a number: "))
2
3 if x < 0:
4     x = 0 # sets x equal to 0 if x was less than 0
5
6 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:

```
1 x = 5
2
3 if x % 2 == 0:
4     print("x is even.")
5 else:
6     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.

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:

```
1 password = "hunter2"
2
3 user_pass = input("Please input the password: ")
4
5 if password == user_pass:
6     print("Password is correct. Welcome!")
7 else:
8     print("Invalid password.")
```

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

```
1 password = "hunter2"
2 also_password = "hunter3"
3 another_password = "hunter4"
4 user_pass = input("Please input the password: ")
5
6 if password == user_pass:
7     print("Welcome, administrator.")
8 elif user_pass == also_password:
```



```
9     print("Welcome, administrator.")
10 elif user_pass == another_password:
11     print("Welcome, manager.")
12 else:
13     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:

```
1 user_pass = input("Please input the password: ")
2
3 if user_pass == "hunter2" or user_pass == "hunter3":
4     print("Welcome, administrator.")
5 elif user_pass == "hunter4":
6     print("Welcome, manager.")
7 else:
8     print("Wrong password.")
```

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

```
1 x = float(input("Enter a value for x: "))
2 y = float(input("Enter a value for y: "))
3
4 if x > 0:
5     if y > 0:
6         print("Both are greater than 0.")
7     else:
8         print("x is greater than 0, but y is smaller or equal to 0")
9 else:
10    print("x is smaller or equal to 0.")
```

3.1 Summary

- Conditional statements look like this:

```
1  if condition:
2      # some code to run
3  elif othercondition:
4      # some other code to run
5  else:
6      # 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:

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

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

```
1 Please input the temperature in Celsius: 10
2 Please choose Kelvin (K) or Fahrenheit (F): E
3 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!

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

```
1 Enter a number to use: 16
2 Which operation? sqrt (s), arcsin (a), arccos (c), arctan (t): a
3 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:

```
1 >>> x = 5
2 >>> print("x is equal to " + str(x))
3 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.

```
1 >>> x = 5
2 >>> print("x is equal to {}".format(x))
3 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.

```
1 >>> x = 5
2 >>> y = 6
3 >>> print("x is equal to {} and y is equal to {}".format(x, y))
4 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.

```
1 >>> import math
2 >>> print(math.pi)
3 3.141592653589793
4 >>> print("{:.2f}".format(math.pi))
5 3.14
```

For more format options, see

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

4.1 Summary

- Syntax:

```
1 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:

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

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

5.2 Nesting

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

```
1 parity = input("Even or odd? ")
2
3 # prints even or odd numbers between 0 and 10, depending on user input
4 if parity == "odd":
5     n = 1
6     while n < 11:
7         if n % 2 == 1:
8             print(n)
9         n = n + 2
10 elif parity == "even":
11     n = 0
12     while n <= 10:
13         print(n)
14         n = n + 2
15 else:
16     print("You did not enter even or odd.")
```

5.3 Summary

- Syntax:

```
1 while condition:
2     # 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.

- There can be infinite while loops.
- You can nest conditional statements and loops any way you want in any combination.

5.4 Exercises

rps.py Write a program that reads a character for playing the game of rock-paper-scissors. If the character entered by the user is not one of “R”, “P”, or “S”, the program keeps on prompting the user to enter a character.

For example:

```
1 Enter R, P, or S >>> A
2 Did not enter R, P, or S. Try again.
3 Enter R, P, or S >>> R
4 You chose rock. Exiting.
```

sums.py Write a program that keeps prompting the user for numbers to add to a sum until the user types in “exit”. Then, display the sum of the numbers previously entered. Assume the user input is nothing other than numbers or “exit”.

For example:

```
1 Enter a number to add to the sum >>> 15
2 Enter a number to add to the sum >>> 14.5
3 Enter a number to add to the sum >>> 12.25
4 Enter a number to add to the sum >>> exit
5 Sum of numbers: 41.75
```

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:

```
1 Enter a number: 16
2 1
3 2
4 3 Fizz
5 4
6 5 Buzz
7 6 Fizz
8 7
9 8
10 9 Fizz
11 10 Buzz
12 11
13 12 Fizz
14 13
15 14
16 15 FizzBuzz
17 16
```

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


primes.py 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`.

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
<code>turtle.forward(x)</code>	move the turtle forward x pixels
<code>turtle.backward(x)</code>	move the turtle backward x pixels
<code>turtle.left(x)</code>	turn the turtle left x degrees
<code>turtle.right(x)</code>	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(60)
8     counter = counter + 1
```

6.1 Exercises

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



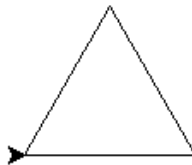
Hint: Try this on paper first. (It does not matter if the turtle is still displayed once it is drawn.)

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
10 Please enter a direction: forward
11 Please enter a direction: forward
12 Please enter a direction: left
13 How many degrees? 45
14 Please enter a direction: left
15 How many degrees? 45
16 Please enter a direction: forward
17 Please enter a direction: right
18 How many degrees? 45
19 Please enter a direction: forward
20 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 5.4)
- primes.py (see Section 5.4)
- rps.py (see Section 5.4)
- sums.py (see Section 5.4)
- polygons.py (see Section 6.1)
- navigate.py (see Section 6.1)
- star.py (see Section 6.1)

You should submit your code as a tarball. It should contain all files used in the exercises for this lab. The submitted file should be named

`cse107_firstname_lastname_lab2.tar.gz`

Upload your tarball to Canvas.