Fundamentals of Programming 1

Lecture 8

Aurelia Power, ITB, 2018

Introduction to Functions

- So far we have designed scripts as a single series of instructions.
- However, maintaining large programs using this approach is unfeasible.
- We can use a different approach: the **divide and conquer** approach - to facilitate a large program to be broken down into smaller and more manageable parts.
- We can take advantage of the fact that most programs, especially large ones, have instructions that can be grouped as a separate set of instructions, each set carrying out specific tasks.
- A group of instructions that accomplishes a specific task is called a **routine**.
- Program routines in Python are called **functions**.

Why Functions?

- There are several motivations for using functions, and we will focus on 3 motives.

Motivation/ Reason 1

Motive 1

The “divide-and-conquer” approach makes **program development very manageable**; that is, using functions makes a complex task more manageable by allowing it to be broken into smaller manageable steps/units.

Remember…

“Take a large problem,
break it into smaller subproblems,
solve the smaller problems and
thereby solve the big problem"

Why Functions?

Motivation/ Reason 2

Motive 2

- Another motivation is **software re-usability**, because we can re-use the functions (built-in or otherwise) without having to create applications from scratch.
- Functions/Routines are <u>the building blocks </u>to create a new programs in Python and they can be called/invoked as many times as needed in a given program.
- We have already used many **built-in functions** that : print, format, range, len, etc.
- We have also used **functions that are found in other modules** of the Python library, such as the *randint* function (from the *random* module).

Why Functions?

Motivation/ Reason 3

Motive 3

- Another motivation is **separation of concerns,** which means that functions can be used over and over again for <u>specific purposes without having to depend on other units.</u>

- This is the case because functions are units that carry out **specific tasks** that deal with very scope-restricted problems**.**

- Each function can be thought of as a <u>black box </u>that you can call/invoke to carry out a task, but you don't need to know/or be concerned with how was implemented (coded).

- REMEMBER: Functions have their own names, definitions and implementations…

What are functions?

- So, what exactly is a function???

A function is a very specific sequence of instructions (a routine) with a meaningful name that is limited to performing a single, well-defined task which can then be called/invoked/used in any program as many times as needed.

- The **name** should express the task that is carried out by that function.
- And we already have encountered some built-in function from the Python library that carry out very specific tasks and which have very telling names:
  - print function which outputs the arguments passed;
  - len function which returns the number of elements a sequence such as a string has;
  - int function which converts a string to an integer value and returns that integer value.

What are functions?

- But we can also write our own functions**: programmer-defined functions**.
  - Like built-in functions, they should also carry out specific tasks, and be called/invoked/used at many points in a program (as needed):

  - A function definition is made up of 2 elements:

1. The **function header**
   1. starts with the keyword **def**
   2. followed by **an identifier** - the **function name**;
   3. the function name is followed by a **comma-separated (possibly empty) list of identifiers** enclosed within parentheses which are **names for the formal parameters;**
   4. The header must end with a **colon** (**:**)

2. The **function body/function block** which contains the instruction(s) to be carried out by the function; the body must be indented.

What are functions?

```
def function_name ( parameter1, parameter2, parameter3 ):      ← header
    instruction(s) to be executed                              ← body
```

- The number of parameters indicates the number of values that must be passed to the function when called/invloked/used - aka **actual arguments**: so the parameters act as placeholders for the arguments passed:

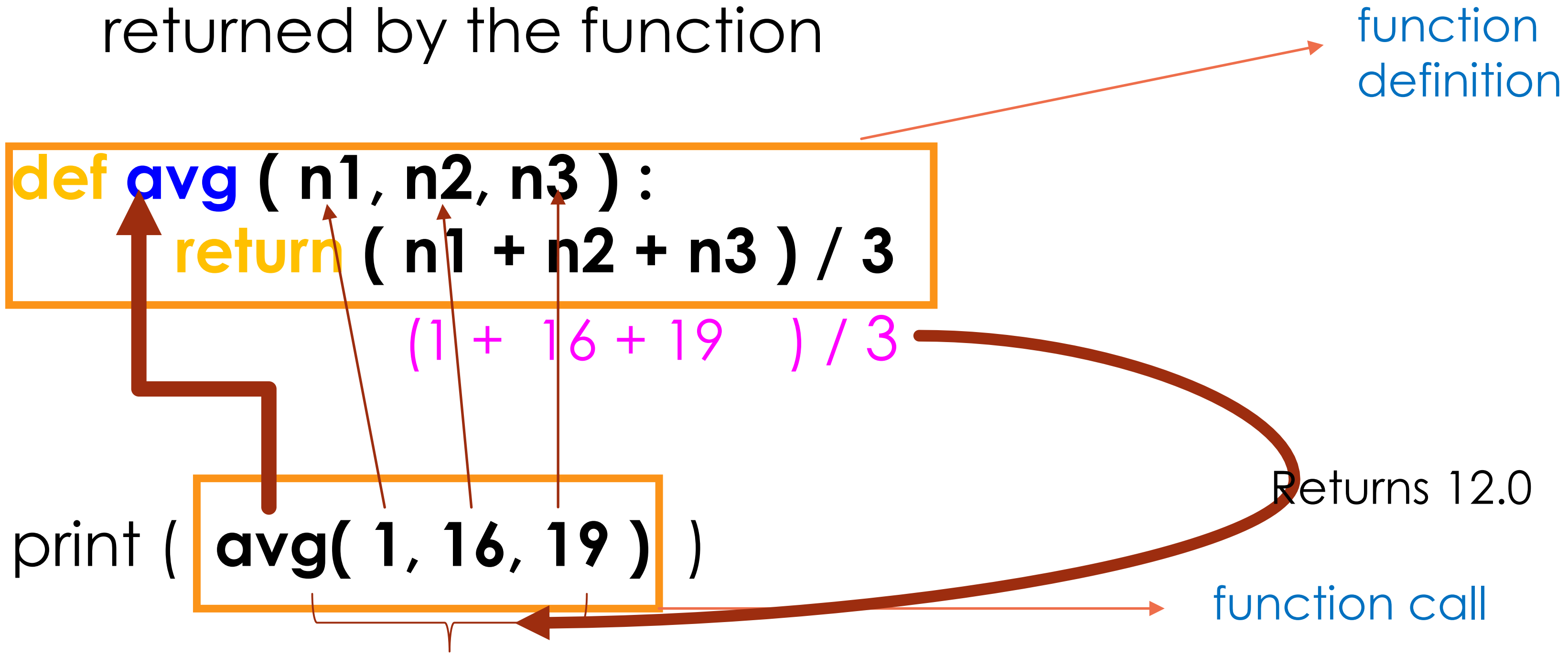
```
function_name(a1, a2, a3) ## the function must be invoked with 3 arguments
## parameter1 will be taking the value of a1, parameter2 will be taking the
## value of n2, and parameter3 will be taking the value of a3
```

- The number of function parameters may vary, depending on what data does the function need in order to carry out its designated task → the parameter list may be empty (no parameters) or it may contain 1 parameter, or 2 parameters, and so on…
- Functions are typically defined at the beginning of a program because they must be defined before they are called/invoked/used.

What are functions?

- In Python we typically have 2 types of functions:

1. **Value returning functions**:
   1. A routine that is called when we need its return value, such as **len**
   2. Similar to mathematical functions: f(x) = 2x – if we plug in any value, its result is that value multiplied by 2
   3. It contains a return statement that specifies the value is to be returned by the function



```
def avg ( n1, n2, n3 ) :
        return ( n1 + n2 + n3 ) / 3

print ( avg( 1, 16, 19 ) )
```

What are functions?

2. Non-Value Returning functions:

1. A routine that is called for its side effects, such as **print**
2. A side effect is an action such as printing/displaying
3. By default, their return value is **None**

```
def sayHello ( ):
        print ( 'Hello world!' )
        print ( 'This is my hello function!' )
```

function definition

```
sayHello ()
```

function call

NOTE: There is a fundamental difference between how value-returning functions and non-value returning functions are called:

- A call to a value returning function is an expression that can be passed to another function, or assigned to a variable, or used to compute some other values…
- A call to a non-value returning function is a statement

Your turn…

- The identifiers between the parentheses when defining a function are called:
  (formal) parameters
- The values passed when a function is called/invoked are called:
  (actual) arguments
- A call to a non-returning value function is a(n):
  statement
- A call to a returning value function is a(n):
  expression
- The default return of a non-retutning function is:
  None

Your turn…

- Write and invoke a function that takes in a name and prints hello to that name:

```
def printHello (name):
      print ('Hello', name)

printHello('John' )
printHello ('Anna')
```

- Write and invoke a function that takes in a number and returns its absolute value:

```
def absoluteVal (n):
      if n < 0:
            return –n
      return n

print( absoluteVal (-3),  absoluteVal (7) )
```

More on formal parameters and actual arguments

- Actual arguments are the values that are passed to a called function's parameters to be operated on;
- Actual arguments will initialise the formal parameters that are specified when defining a function.
- The <u>correspondence of the actual arguments and formal parameters is determined by the order of arguments passed</u>, and not by their names:

```
def ordered_numbers (n1, n2):
        return n1 < n2

a = int(input("enter your age: "))
b = int(input("enter your brother's age: "))

if ordered_numbers (a, b):
    print("you are younger...")
if ordered_numbers (b, a):
    print("your brother is younger")
```

n1 and n2 are formal parameters

a and b are actual arguments → n1 will take the value of a, and n2 the value of b

b and a are actual arguments → n1 will take the value of b this time, and n2 the value of a

Positional Arguments in Python

- So far, we have looked at functions with a fixed number of positional arguments.
- A **positional argument** is an argument that is assigned to a particular parameter based on its position in the argument list:

```
def ordered_numbers (n1, n2):
        return n < n2

isLess = ordered_numbers( 25, 12 )
```

Defining the function ordered_numbers

Calling/Invoking/Using the function ordered_numbers

```
def print_n_times ( s, n ):
        for x in range(n):
            print(s, end=‘  ’)

print_n_times ( ‘hello’, 7)
```

Defining the function print_n_times

Calling/Invoking/Using the function print_n_times

Keyword Arguments in Python

- In Python we can call a function by the use of keyword arguments, and in this case, the position of the arguments does not matter.
- A **keyword argument** is an argument that is assigned to a particular parameter based on the name of the parameter:

```
def ordered_numbers (n1, n2):
        return n1 < n2

isLess = ordered_numbers(n2=25, n1=12 )
```

Defining the function ordered_numbers

Calling/Invoking/Using the function ordered_numbers

```
def print_n_times ( s, n ):
        for x in range(n):
            print(s, end=‘  ’)

print_n_times ( s=‘hello’, n=7)
```

Defining the function print_n_times

Calling/Invoking/Using the function print_n_times

- Useful when it’s easier to remember the parameter name rather than its position

Default Arguments in Python

- In Python we can also assign a default value to any parameter, and in that case, its value must be specified when defining the function.
- Then, a **default argument** is an argument that is optional, and when omitted it will take the default value:

```
def ordered_numbers (n1=0, n2=0):
        return n1 < n2

print( ordered_numbers() )

print( ordered_numbers(n1=21) )

print( ordered_numbers( n2=3) )

print(ordered_numbers(7, 19))

print(ordered_numbers(7)) ##??
```

Defining the function ordered_numbers

Calling the function with no arguments, so both parameters will take the default values of 0

Calling the function with one keyword argument, so n1 will take the value of 21, and n2 will take the default value of 0

Calling the function with one keyword argument, so n1 will take the default value of 0, and n2 will take the value of 3

Calling the function with positional arguments, so n1 will take the value of 7, and n2 will take the value of 19

Your turn…

- Consider the following function definition:

```
def print_n_times ( s='python', n=1 ):
    for x in range(n):
        print(s, end=‘ ‘)
```

- What will each of the following calls produce?

```
print_n_times ( s=‘hello', n=-1) ##nothing
print_n_times ( ) ## python
print_n_times ( n=3) ## python python python
print_n_times ( s='anaconda’) ## anaconda
print_n_times ( n='anaconda’) ## TypeError
print_n_times ( s=5) ## 5
print_n_times ('viper', 2) ## viper viper
```

Your turn…

- Define a function that counts down to a bomb explosion after a certain amount of seconds; for example, if we passed 5 seconds, it should print 5 4 3 2 1 BOOM!!!; if we passed 2 seconds, it should print 2 1 BOOM!!!:

```
def  explode_bomb(n):
   while n >= 0:
        if  n != 0:
            print(n, end=‘ ‘)
        else:
            print(‘BOOM!!!’)
        n -= 1
```

- To the above function add another parameter for what you want to output instead of ‘BOOM!!!’
- Modify the above function so that parameters will take some default values.
- Call the function several times using positional arguments, keyword arguments and default arguments.

More on Parameter Passing… Mutable vs. Immutable Arguments

- Parameter passing = the process of calling a function using arguments.
- As already said, actual arguments are the values that will initialise the formal parameters when the function is called, so that it will use those values to carry out its designated task.

For example:

```
def increase_by_1 ( n ):
    return (n + 1)
x = 7
print ( increase_by_1( x) )  ## outputs 8
```

- But… does the value of x changes to 8???
- No… because integer values are immutable → x still points to 7

```
print (x ) ## outputs 7
```

- Similarly, if you pass as arguments floats, Booleans, and strings they are not altered, no matter what the task of the function is.

Your turn…

- Consider the following function definitions:

```
def some_operation ( s1, s2 ):
    return s1 + s2
def other_operation (n1, n2):
    return n1 * n2
```

- What do the following function calls output??

```
some_operation (“spider”, “web”)
other_operation (3, 7)
x = ‘3’; y = ‘7’
some_operation(x, y); other_operation (int(x), int(y))
print(other_operation (x, y)); print(some_operation (3, 4))
print(x, y)
print(some_operation (“foot”, 3)); print(other_operation( int(x), ‘7’ ) )
```

Temperature conversion example with appropriate functions…

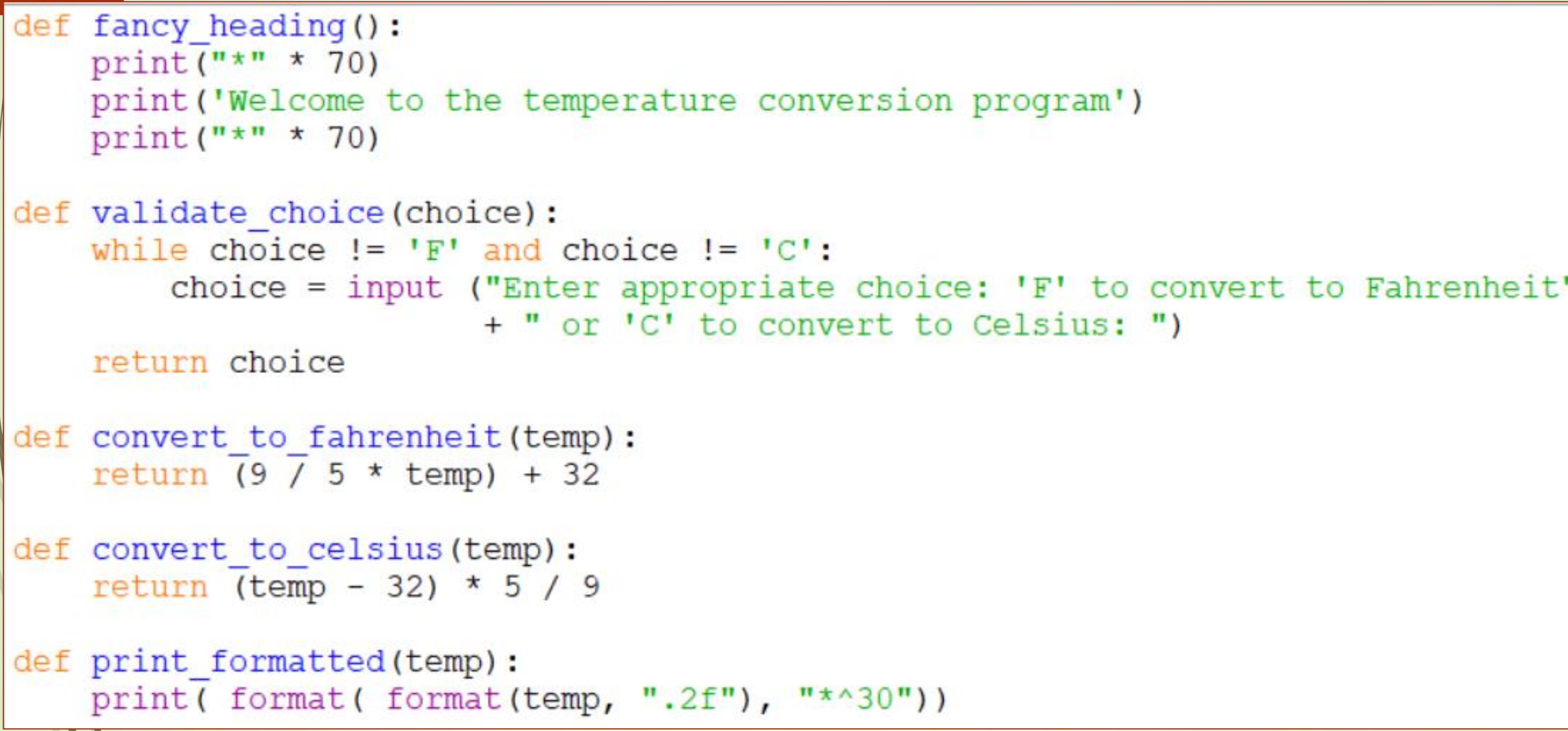
- The program must allow the user to enter **F** or **C** to convert a series of temperatures to either **Fahrenheit** or to **Celsius**;
- The program should also define functions to carry out the following tasks:

1. Displaying some fancy heading that also specifies the purpose of the script
2. Validating the user's choice to ensure that it is either the character F or the character C; if it is something else, the user should be prompted repeatedly to re-enter either F or C
3. Convert the temperature to Fahrenheit and another to Celsius
4. Printing formatted temperatures with 2 decimal places and centered in a field of 30

PSEUDCODE:

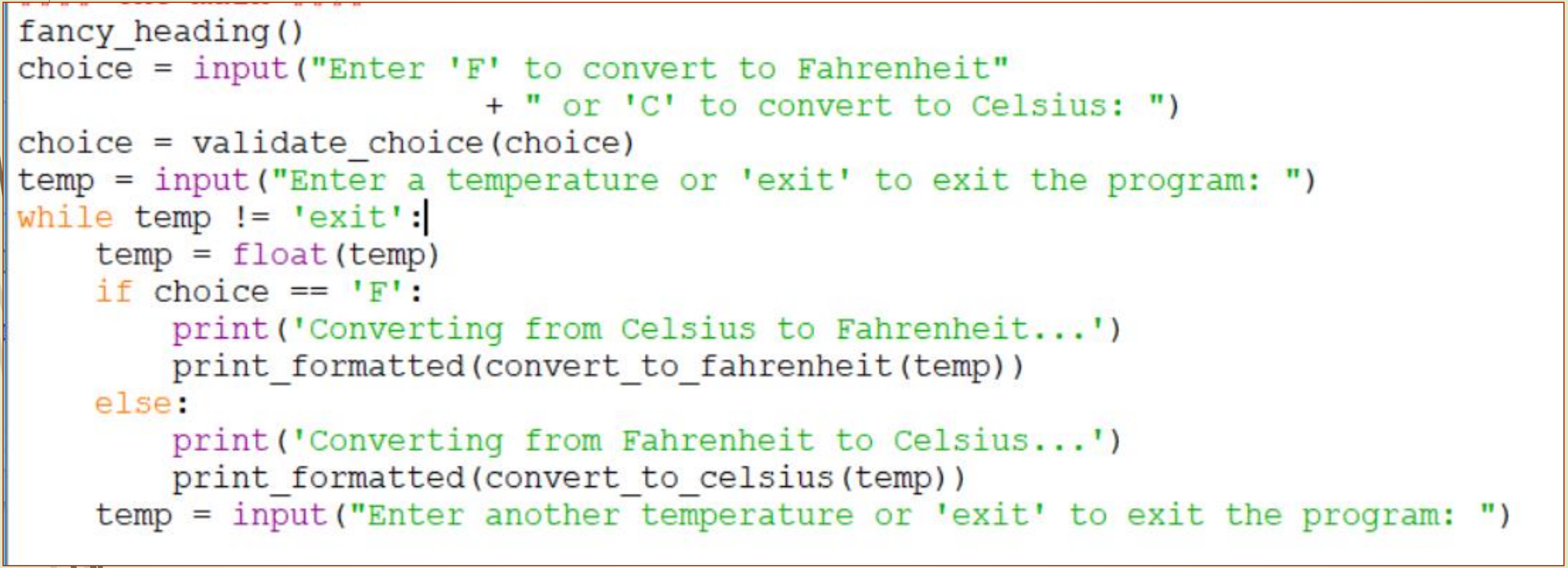
```
- Define the heading function
- Define the input validation function
- Define the 2 conversion functions
- Call the heading function
- Prompt user to enter a series of temperatures or 'exit' to exit the program
    - Validate the input
    - Check to see whether it is F or C
        - If it is F call the Fahrenheit conversion function
        - Else call the Celsius conversion function
```

The Functions …



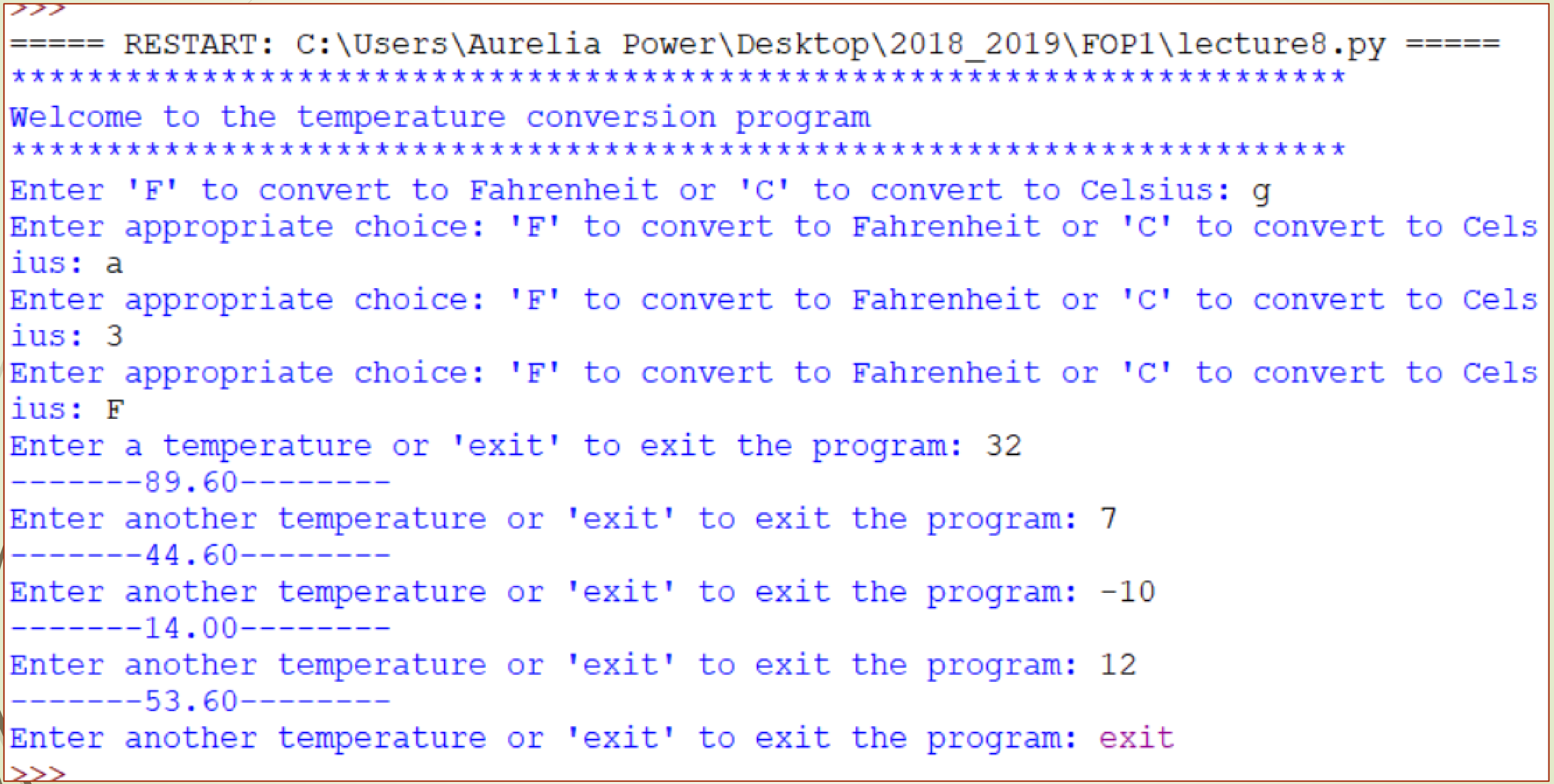
```
def fancy_heading():
    print("*" * 70)
    print('Welcome to the temperature conversion program')
    print("*" * 70)

def validate_choice(choice):
    while choice != 'F' and choice != 'C':
        choice = input ("Enter appropriate choice: 'F' to convert to Fahrenheit'
                        + " or 'C' to convert to Celsius: ")
    return choice

def convert_to_fahrenheit(temp):
    return (9 / 5 * temp) + 32

def convert_to_celsius(temp):
    return (temp - 32) * 5 / 9

def print_formatted(temp):
    print( format( format(temp, ".2f"), "*^30"))
```

Calling the functions…



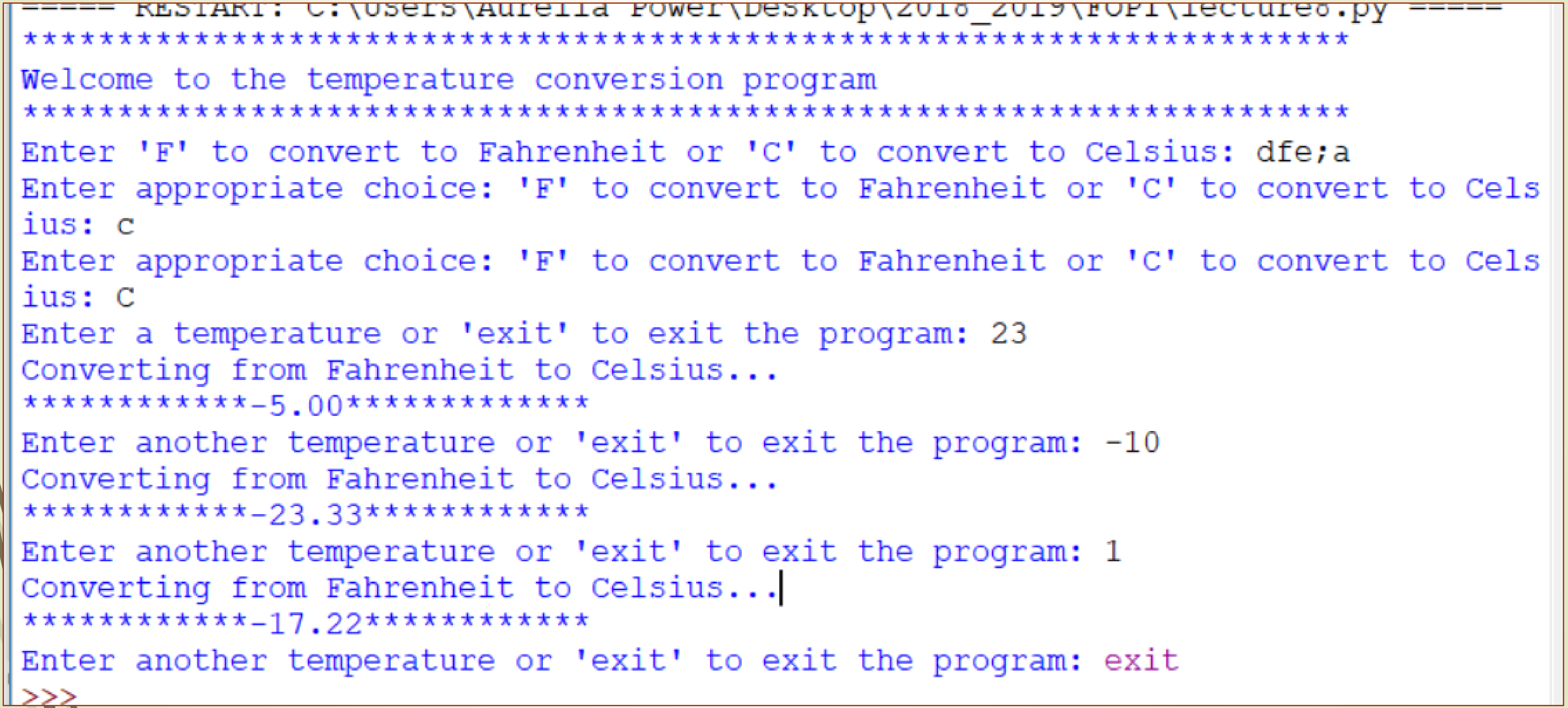
```
fancy_heading()
choice = input("Enter 'F' to convert to Fahrenheit"
                     + " or 'C' to convert to Celsius: ")
choice = validate_choice(choice)
temp = input("Enter a temperature or 'exit' to exit the program: ")
while temp != 'exit':
    temp = float(temp)
    if choice == 'F':
        print('Converting from Celsius to Fahrenheit...')
        print_formatted(convert_to_fahrenheit(temp))
    else:
        print('Converting from Fahrenheit to Celsius...')
        print_formatted(convert_to_celsius(temp))
    temp = input("Enter another temperature or 'exit' to exit the program: ")
```

Running the program … choosing ‘F’



```
>>>
===== RESTART: C:\Users\Aurelia Power\Desktop\2018_2019\FOP1\lecture8.py =====
**********************************************************************
Welcome to the temperature conversion program
**********************************************************************
Enter 'F' to convert to Fahrenheit or 'C' to convert to Celsius: g
Enter appropriate choice: 'F' to convert to Fahrenheit or 'C' to convert to Cels
ius: a
Enter appropriate choice: 'F' to convert to Fahrenheit or 'C' to convert to Cels
ius: 3
Enter appropriate choice: 'F' to convert to Fahrenheit or 'C' to convert to Cels
ius: F
Enter a temperature or 'exit' to exit the program: 32
-------89.60--------
Enter another temperature or 'exit' to exit the program: 7
-------44.60--------
Enter another temperature or 'exit' to exit the program: -10
-------14.00--------
Enter another temperature or 'exit' to exit the program: 12
-------53.60--------
Enter another temperature or 'exit' to exit the program: exit
>>>
```

Running the program … choosing ‘C



```
===== RESTART: C:\Users\Aurelia Power\Desktop\2018_2019\FOP1\lecture8.py =====
*********************************************************************
Welcome to the temperature conversion program
*********************************************************************
Enter 'F' to convert to Fahrenheit or 'C' to convert to Celsius: dfe;a
Enter appropriate choice: 'F' to convert to Fahrenheit or 'C' to convert to Cels
ius: c
Enter appropriate choice: 'F' to convert to Fahrenheit or 'C' to convert to Cels
ius: C
Enter a temperature or 'exit' to exit the program: 23
Converting from Fahrenheit to Celsius...
************-5.00*************
Enter another temperature or 'exit' to exit the program: -10
Converting from Fahrenheit to Celsius...
************-23.33************
Enter another temperature or 'exit' to exit the program: 1
Converting from Fahrenheit to Celsius...
************-17.22************
Enter another temperature or 'exit' to exit the program: exit
>>>
```

Your Turn…

- Write a program that allows us to the gpa for the current year; the program should define the following functions:

1. To display some nice heading
2. To validate that the grade entered is one of the following options: A, B+, B, B-, C+, C, D, F
3. To calculate the gpa using the six first semester modules, each worth 5 credits
4. To format and print the gpa with 1 decimal place and right justified in a field of 20 characters

- The program should repeatedly prompt the user to enter six grades, that need to be checked that they are valid
- Once 6 valid grades are entered, the gpa should be calculated and displayed