11 Prepare: Functional Programming

A **paradigm** is a way of thinking or a way of perceiving the world. There are at least four main paradigms for programming a computer: procedural, declarative, functional, and object-oriented. During previous lessons in CSE 110 and 111, you used procedural programming. During this lesson, you will be introduced to functional programming.

Procedural programming is a programming paradigm that focuses on the process or the steps to accomplish a task. This is the type of programming that you did in CSE 110 and in previous lessons of CSE 111.

Declarative programming is a programming paradigm that does *not* focus on the process or steps to accomplish a task. Instead, with declarative programming, a programmer focuses on what she wants from the computer, or in other words, she focuses on the desired results. The SQL programming language is a good example of a declarative language. If you have ever written SQL code, then you have used declarative programming. When writing SQL code, a programmer writes code to tell the computer what she wants in the results but not the steps the computer must follow to get those results.

Functional programming is a programming paradigm that focuses on functions and avoids shared state, mutating state, and side effects. There are many techniques and concepts that are part of functional programming. However, in this lesson we will focus on just three, namely:

1. We can pass a function into another function.
2. A nested function is a function defined inside another function.
3. A lambda function is a small anonymous function.

Concepts

Here are the functional programming concepts that you should learn during this lesson.

Passing a Function into another Function

The Python programming language allows a programmer to pass a function as an argument into another function. A function that accepts other functions in its parameters is known as a **higher-order function**. Higher-order functions are often used to process the elements in a list. Before seeing an example of using a higher-order function to process a list, first consider the program in example 1 that doesn't use a higher-order function but instead uses a `for` loop to convert a list of temperatures from Fahrenheit to Celsius.

```python
# Example 1

def main():
    fahr_temps = [72, 65, 71, 75, 82, 87, 68]
```

```
 5
 6      # Print the Fahrenheit temperatures.
 7      print(f"Fahrenheit: {fahr_temps}")
 8
 9      # Convert each Fahrenheit temperature to Celsius and store
10      # the Celsius temperatures in a list named cels_temps.
11      cels_temps = []
12      for fahr in fahr_temps:
13          cels = cels_from_fahr(fahr)
14          cels_temps.append(cels)
15
16      # Print the Celsius temperatures.
17      print(f"Celsius: {cels_temps}")
18
19
20  def cels_from_fahr(fahr):
21      """Convert a Fahrenheit temperature to
22      Celsius and return the Celsius temperature.
23      """
24      cels = (fahr - 32) * 5 / 9
25      return round(cels, 1)
26
27
28  # Call main to start this program.
29  if __name__ == "__main__":
30      main()
```

```
> python example_1.py
Fahrenheit: [72, 65, 71, 75, 82, 87, 68]
Celsius: [22.2, 18.3, 21.7, 23.9, 27.8, 30.6, 20.0]
```

At lines 12–14 in example 1, there is a `for` loop that converts each Fahrenheit temperature to Celsius and then appends the Celsius temperature onto a new list. Writing a `for` loop like this is the traditional way to process all the elements in a list and doesn't use higher-order functions.

Python includes a built-in higher-order function named `map` that will process all the elements in a list and return a new list that contains the results. The map function accepts a function and a list as arguments and contains a loop inside it, so that when a programmer calls the `map` function, he doesn't need to write a loop. The `map` function is a higher-order function because it accepts a function as an argument. Consider the program in example 2 that produces the same results as example 1.

```
 1  # Example 2
 2
 3  def main():
 4      fahr_temps = [72, 65, 71, 75, 82, 87, 68]
 5
 6      # Print the Fahrenheit temperatures.
 7      print(f"Fahrenheit: {fahr_temps}")
 8
 9      # Convert each Fahrenheit temperature to Celsius and store
10      # the Celsius temperatures in a list named cels_temps.
11      cels_temps = list(map(cels_from_fahr, fahr_temps))
12
13      # Print the Celsius temperatures.
14      print(f"Celsius: {cels_temps}")
15
16
17  def cels_from_fahr(fahr):
```

```
18      """Convert a Fahrenheit temperature to
19      Celsius and return the Celsius temperature.
20      """
21      cels = (fahr - 32) * 5 / 9
22      return round(cels, 1)
23
24
25  # Call main to start this program.
26  if __name__ == "__main__":
27      main()
```

```
> python example_2.py
Fahrenheit: [72, 65, 71, 75, 82, 87, 68]
Celsius: [22.2, 18.3, 21.7, 23.9, 27.8, 30.6, 20.0]
```

Notice that example 2, doesn't contain a `for` loop. Instead, at line 11, it contains a call to the `map` function. Remember that the `map` function has a loop inside it, so that the programmer who calls `map`, doesn't have to write the loop. Notice also at line 11 that the first argument to the `map` function is the name of the `cels_from_fahr` function. In other words, at line 11, we are passing the `cels_from_fahr` function into the `map` function, so that `map` will call `cels_from_fahr` for each element in the *fahr_temps* list.

The `map` function is just one example of a higher-order function. Python also includes the built-in higher-order `sorted` and `filter` functions and several higher-order functions in the `functools` module.

Nested Functions

The Python programming language allows a programmer to define nested functions. A **nested function** is a function that is defined inside another function and is useful when we wish to split a large function into smaller functions and the smaller functions will be called by the containing function only. The program in example 3 produces the same results as examples 1 and 2, but it uses a nested function. Notice in example 3 at lines 5–10 that the `cels_from_fahr` function is nested inside the `main` function.

```
 1  # Example 3
 2
 3  def main():
 4
 5      def cels_from_fahr(fahr):
 6          """Convert a Fahrenheit temperature to
 7          Celsius and return the Celsius temperature.
 8          """
 9          cels = (fahr - 32) * 5 / 9
10          return round(cels, 1)
11
12      fahr_temps = [72, 65, 71, 75, 82, 87, 68]
13
14      # Print the Fahrenheit temperatures.
15      print(f"Fahrenheit: {fahr_temps}")
16
17      # Convert each Fahrenheit temperature to Celsius and store
18      # the Celsius temperatures in a list named cels_temps.
19      cels_temps = list(map(cels_from_fahr, fahr_temps))
20
21      # Print the Celsius temperatures.
```

```python
    print(f"Celsius: {cels_temps}")


# Call main to start this program.
if __name__ == "__main__":
    main()
```

```
> python example_3.py
Fahrenheit: [72, 65, 71, 75, 82, 87, 68]
Celsius: [22.2, 18.3, 21.7, 23.9, 27.8, 30.6, 20.0]
```

Lambda Functions

A Python **lambda function** is a small anonymous function, meaning a small function without a name. A lambda function is always a small function because the Python language restricts a lambda function to just one expression. Consider the program in example 4 which is yet another example program that converts Fahrenheit temperatures to Celsius. Notice the lambda function at line 12 of example 4. It takes one parameter named *fahr* and computes and returns the corresponding Celsius temperature. At line 16, the lambda function is passed into the `map` function.

```python
 1  # Example 4
 2
 3  def main():
 4      fahr_temps = [72, 65, 71, 75, 82, 87, 68]
 5
 6      # Print the Fahrenheit temperatures.
 7      print(f"Fahrenheit: {fahr_temps}")
 8
 9      # Define a lambda function that converts
10      # a Fahrenheit temperature to Celsius and
11      # returns the Celsius temperature.
12      cels_from_fahr = lambda fahr: round((fahr - 32) * 5 / 9, 1)
13
14      # Convert each Fahrenheit temperature to Celsius and store
15      # the Celsius temperatures in a list named cels_temps.
16      cels_temps = list(map(cels_from_fahr, fahr_temps))
17
18      # Print the Celsius temperatures.
19      print(f"Celsius: {cels_temps}")
20
21
22  # Call main to start this program.
23  if __name__ == "__main__":
24      main()
```

```
> python example_4.py
Fahrenheit: [72, 65, 71, 75, 82, 87, 68]
Celsius: [22.2, 18.3, 21.7, 23.9, 27.8, 30.6, 20.0]
```

Some students are confused by the statement that a lambda function is an anonymous function (a function without a name). Looking at the lambda function in example 4 at line 12, it appears that the lambda function is named *cels_from_fahr*. However, *cels_from_fahr* is the name of a variable, not the name of the lambda function. The lambda function has no name. This distinction may seem trivial until we see an example of an inline lambda function. Notice in the next example that the lambda function is defined inside the parentheses for the call to the `map` function.

```
# Convert each Fahrenheit temperature to Celsius and store
# the Celsius temperatures in a list named cels_temps.
cels_temps = list(map(
        lambda fahr: round((fahr - 32) * 5 / 9, 1),
        fahr_temps))
```

To write a lambda function write code that follows this template:

```
lambda param1, param2, … paramN: expression
```

As shown in the template, type the keyword `lambda`, then parameters separated by commas, then a colon (:), and finally an expression that performs arithmetic, modifies a string, or computes something else.

In Python, every lambda function can be written as a regular Python function. For example, the lambda function in example 4 can be rewritten as the `cels_from_fahr` function in examples 1, 2, and 3.

Example - Map and Filter

The checkpoint for lesson 9 required you to write a program that replaced all the occurrences of "AB" in a list with the name "Alberta" and then counted how many times the name "Alberta" appeared in the list.

Example 5 contains a program that uses the `map` and `filter` functions to complete the requirements of the lesson 9 checkpoint. The example program works by doing the following:

1. Calling the `read_list` function at line 6 to read all the provinces from a text file into a list. (The `read_list` function is in the preparation content for lesson 9.)
2. Calling the `map` function at line 22 to convert all elements that are "AB" to "Alberta."
3. Calling the `filter` function at line 34 to remove all elements that are not "Alberta."
4. Calling the `len` function at line 42 to count the number of elements that remain in the filtered list.

```
 1  # Example 5
 2
 3  def main():
 4      # Read a file that contains a list
 5      # of Canadian province names.
 6      provinces_list = read_list("provinces.txt")
 7
 8      # As a debugging aid, print the entire list.
 9      print("Original list of provinces:")
10      print(provinces_list)
11      print()
12
13      # Define a nested function that converts AB to Alberta.
```

```
    def alberta_from_ab(province_name):
        if province_name == "AB":
            province_name = "Alberta"
        return province_name

    # Replace all occurrences of "AB" with "Alberta" by
    # calling the map function and passing the ablerta_from_ab
    # function and provinces_list into the map function.
    new_list = list(map(alberta_from_ab, provinces_list))
    print("List of provinces after AB was changed to Alberta:")
    print(new_list)
    print()

    # Define a lambda function that returns True if a
    # province's name is Alberta and returns False otherwise.
    is_alberta = lambda name: name == "Alberta"

    # Filter the new list to only those provinces that
    # are "Alberta" by calling the filter function and
    # passing the is_alberta function and new_list.
    filtered_list = list(filter(is_alberta, new_list))
    print("List filtered to Alberta only:")
    print(filtered_list)
    print()

    # Because all the elements in filtered_list are
    # "Alberta", we can count how many elements are
    # "Alberta" by simply calling the len function.
    count = len(filtered_list)

    print(f"Alberta occurs {count} times in the modified list.")


# Call main to start this program.
if __name__ == "__main__":
    main()
```

```
> python example_5.py
Original list of provinces:
['Alberta', 'Ontario', 'Prince Edward Island', 'Ontario', 'Quebec',
'Saskatchewan', 'AB', 'Nova Scotia', 'Alberta',
'Northwest Territories', 'Saskatchewan', 'Nunavut', 'Nova Scotia',
'Prince Edward Island', 'Alberta', 'Nova Scotia', 'Nova Scotia',
'Prince Edward Island', 'British Columbia', 'Ontario', 'Ontario',
'Newfoundland and Labrador', 'Ontario', 'Ontario', 'Saskatchewan',
'Nova Scotia', 'Prince Edward Island', 'Saskatchewan', 'Ontario',
'Newfoundland and Labrador', 'Ontario', 'British Columbia',
'Manitoba', 'Ontario', 'Alberta', 'Saskatchewan', 'Ontario',
'Yukon', 'Ontario', 'New Brunswick', 'British Columbia',
'Manitoba', 'Yukon', 'British Columbia', 'Manitoba', 'Yukon',
'Newfoundland and Labrador', 'Ontario', 'Yukon', 'Ontario',
'AB', 'Nova Scotia', 'Newfoundland and Labrador', 'Yukon',
'Nunavut', 'Northwest Territories', 'Nunavut', 'Yukon',
'British Columbia', 'Ontario', 'AB', 'Saskatchewan',
'Prince Edward Island', 'Saskatchewan', 'Prince Edward Island',
'Alberta', 'Ontario', 'Alberta', 'Manitoba', 'AB',
'British Columbia', 'Alberta']

List of provinces after AB was changed to Alberta:
['Alberta', 'Ontario', 'Prince Edward Island', 'Ontario', 'Quebec',
'Saskatchewan', 'Alberta', 'Nova Scotia', 'Alberta',
'Northwest Territories', 'Saskatchewan', 'Nunavut', 'Nova Scotia',
'Prince Edward Island', 'Alberta', 'Nova Scotia', 'Nova Scotia',
'Prince Edward Island', 'British Columbia', 'Ontario', 'Ontario',
'Newfoundland and Labrador', 'Ontario', 'Ontario', 'Saskatchewan',
'Nova Scotia', 'Prince Edward Island', 'Saskatchewan', 'Ontario',
'Newfoundland and Labrador', 'Ontario', 'British Columbia',
'Manitoba', 'Ontario', 'Alberta', 'Saskatchewan', 'Ontario',
'Yukon', 'Ontario', 'New Brunswick', 'British Columbia',
'Manitoba', 'Yukon', 'British Columbia', 'Manitoba', 'Yukon',
'Newfoundland and Labrador', 'Ontario', 'Yukon', 'Ontario',
'Alberta', 'Nova Scotia', 'Newfoundland and Labrador', 'Yukon',
'Nunavut', 'Northwest Territories', 'Nunavut', 'Yukon',
'British Columbia', 'Ontario', 'Alberta', 'Saskatchewan',
'Prince Edward Island', 'Saskatchewan', 'Prince Edward Island',
'Alberta', 'Ontario', 'Alberta', 'Manitoba', 'Alberta',
'British Columbia', 'Alberta']

List filtered to Alberta only:
['Alberta', 'Alberta', 'Alberta', 'Alberta', 'Alberta', 'Alberta',
'Alberta', 'Alberta', 'Alberta', 'Alberta', 'Alberta']

Alberta occurs 11 times in the modified list.
```

Example - Sorting a Compound List

Python includes a built-in higher-order function named `sorted` that accepts a list as an argument and returns a new sorted list. Calling the `sorted` function is straightforward for a simple list such as a list of strings or a list of numbers as shown in example 6 and its output.

```python
# Example 6

def main():
    # Create a list that contains country names
    # and print the list.
    countries = [
        "Canada", "France", "Ghana", "Brazil", "Japan"
    ]
    print(countries)

    # Sort the list. Then print the sorted list.
    sorted_list = sorted(countries)
    print(sorted_list)


# Call main to start this program.
if __name__ == "__main__":
    main()
```

```
> python countries.py
['Mexico', 'France', 'Ghana', 'Brazil', 'Japan']
['Brazil', 'France', 'Ghana', 'Japan', 'Mexico']
```

A **compound list** is a list that contains lists. Sorting a compound list is more complex than sorting a simple list. Consider this compound list that contains data about some countries.

```python
    # Create a list that contains data about countries.
    countries = [
        # [country_name, land_area, population, gdp_per_capita]
        ["Mexico", 1972550, 126014024, 21362],
        ["France",  640679,  67399000, 45454],
        ["Ghana",   239567,  31072940,  7343],
        ["Brazil", 8515767, 210147125, 14563],
        ["Japan",   377975, 125480000, 41634]
    ]
```

Perhaps we want the *countries* compound list sorted by country name or perhaps we want it sorted by population. The element that we want a list sorted by is known as the **key element**. If we want to use the `sorted` function to sort a compound list, we must tell the `sorted` function which element is the key element, which we do by passing a small function as an argument into the `sorted` function. This small function is called the **key function** and extracts the key element from a list as shown in example 7.

Notice at line 26 in example 7, there is a lambda function that extracts the population from a country. Then at line 29 that lambda function is passed to the `sorted` function so that the `sorted` function will sort the list of countries by the population.

```
 1  # Example 7
 2  
 3  def main():
 4      # Create a list that contains data about countries.
 5      countries = [
 6          # [country_name, land_area, population, gdp_per_capita]
 7          ["Mexico", 1972550, 126014024, 21362],
 8          ["France",  640679,  67399000, 45454],
 9          ["Ghana",   239567,  31072940,  7343],
10          ["Brazil", 8515767, 210147125, 14563],
11          ["Japan",   377975, 125480000, 41634]
12      ]
13  
14      # Print the unsorted list.
15      print("Original unsorted list of countries")
16      for country in countries:
17          print(country)
18      print()
19  
20      # Define a lambda function that will be used as the
21      # key function by the sorted function. The lambda
22      # function extracts the population data from a
23      # country so that the population will be used for
24      # sorting the list of countries.
25      POPULATION_INDEX = 2
26      popul_func = lambda country: country[POPULATION_INDEX]
27  
28      # Sort the list of countries by the population.
29      sorted_list = sorted(countries, key=popul_func)
30  
31      # Print the sorted list.
32      print("List of countries sorted by population")
33      for country in sorted_list:
34          print(country)
35  
36  
37  # Call main to start this program.
38  if __name__ == "__main__":
39      main()
```

```
> python countries.py
Original unsorted list of countries
['Mexico', 1972550, 126014024, 21362]
['France', 640679, 67399000, 45454]
```

```
['Ghana', 239567, 31072940, 7343]
['Brazil', 8515767, 210147125, 14563]
['Japan', 377975, 125480000, 41634]

List of countries sorted by population
['Ghana', 239567, 31072940, 7343]
['France', 640679, 67399000, 45454]
['Japan', 377975, 125480000, 41634]
['Mexico', 1972550, 126014024, 21362]
['Brazil', 8515767, 210147125, 14563]
```

By using a key function it's possible to sort a compound list with a key element that isn't in the list. Consider the compound list named *students* that contains data about various students in example 8. Within the list, each student's given name and surname are stored separately. It is common for a user to want such a list to be sorted by surname and then by given name. A simple way to do that is to write a key function that combines the surname and given name elements and returns the combined name as the key that the `sorted` function will use for sorting.

Lines 21–22 in example 8 contain a `lambda` function that combines a student's surname and given name into a string that is used as the key by the sorted function at line 25. Notice in the output from example 8 that the students are sorted by surname and then by given name.

```
 1  # Example 8
 2
 3  def main():
 4      # Create a list that contains data about young students.
 5      students = [
 6          # [given_name, surname, reading_level]
 7          ["Robert", "Smith", 6.7],
 8          ["Annie", "Smith", 6.2],
 9          ["Robert", "Lopez", 7.1],
10          ["Sean", "Li", 5.6],
11          ["Sofia", "Lopez", 5.3],
12          ["Lily", "Harris", 6.7],
13          ["Alex", "Harris", 5.8]
14      ]
15
16      GIVEN_INDEX = 0
17      SURNAME_INDEX = 1
18
19      # Define a lambda function that combines
20      # a student's surname and given name.
21      combine_names = lambda student_list: \
22          f"{student_list[SURNAME_INDEX]}, {student_list[GIVEN_INDEX]}
23
24      # Sort the list by the combined key of surname, given_name.
25      sorted_list = sorted(students, key=combine_names)
26
27      # Print the list.
28      for student in sorted_list:
29          print(student)
30
31
32  # Call main to start this program.
33  if __name__ == "__main__":
34      main()
```

```
> python students.py
['Alex', 'Harris', 5.8]
['Lily', 'Harris', 6.7]
['Sean', 'Li', 5.6]
['Robert', 'Lopez', 7.1]
['Sofia', 'Lopez', 5.3]
['Annie', 'Smith', 6.2]
['Robert', 'Smith', 6.7]
```

Videos

If you wish to learn more about functional programming in Python, watch these videos by Dan Bader.

- The Basics of Functional Programming in Python (19 minutes)
- The Python `filter` function (16 minutes)
- The Python `map` function (14 minutes)
- The Python `reduce` function (18 minutes)

Additional Documentation

If you wish to learn even more details about functional programming, the following articles contain reference documentation for functional programming in Python.

- Python `map` function
- Python `filter` function
- Python `reduce` function
- Python `functools` module
- A thorough tutorial about lambda functions

Summary

In this preparation content, you learned that functional programming is a programming paradigm that focuses on functions, and you learned these three concepts that are used in functional programming:

1. You can pass a function as an argument into another function.
2. A nested function is a function defined inside another function.
3. A lambda function is a small anonymous function.