SI 506 Lecture 10

Topics

1. Function basics
2. Calling a function inside a loop
3. Functions calling other functions
4. Challenges

Vocabulary

- **Argument**. A value passed to a function or method that corresponds to a parameter defined for the function or method.
- **Caller**. The initiator of a function call.
- **Function**. A defined block of code that performs (ideally) a single task. Functions only run when they are explicitly called. A function can be defined with one or more *parameters* that allow it to accept *arguments* from the caller in order to perform a computation. A function can also be designed to return a computed value. Functions are considered "first-class" objects in the Python eco-system.
- **Parameter**. A named entity in a function or method definition that specifies an argument that the function or method accepts.
- **Scope**. The part of a script or program in which a variable and the object to which it is assigned is visible and accessible.

1.0 Function basics

Writing a function in order to perform a particular task is a form of code modularization designed to simplify otherwise complex processes. Functions also encourage code re-use and adherance to the *Don’t Repeat Yourself* (DRY) principle of software development.

1.1 Defining and calling a function

A function is defined using the keyword `def` and given a name that ends with an open/close parentheses `()`.

In the example below the function named `print_slogan` is called by name and responds by calling the built-in `print()` function and passing it a hard-coded string to print to the screen.

```
def print_slogan():
    print('\n1.1 Snap, Crackle, Pop') # Kellogg's Rice Crispies slogan

print_slogan() # call function
```

❗ A function's indented code block is executed if and only if the function is called explicitly; otherwise, the code is ignored by the Python Interpreter.

1.2 Defining and calling a function with a parameter and return value

Functions are typically defined with one or more *parameters* for accepting input along with a return statement that signals the end of the computation and the "return" of a value to the caller.

❗ A function call is an *expression* (i.e., the call resolves to a value). Functions that do not include a `return` statement return `None` to the caller.

In the example below the function named `print_slogan` is defined with a single parameter named `slogan`. The function accepts input and returns a formatted string.

```
def print_slogan(slogan):
    print(f"\n1.2 {slogan}")

slogan = 'They’rrrre GR-R-REAT'
print_slogan(slogan)
```

1.3 Defining a function with multiple parameters

A function can be specified with multiple parameters. The caller *must* pass to the function the required number of *arguments* or a runtime error will occur.

In the example below, the function `format_slogan` is called with the required arguments passed *by position*. The function's return value is then assigned to a variable.

```
def format_slogan(name, slogan):
    return f"{name} slogan: {slogan}"

cereal = 'Wheaties',
slogan = 'The Breakfast of Champions.'
wheaties_slogan = format_slogan(cereal, slogan) # positional arguments
```

1.4 Positional arguments (order matters)

If positional arguments are passed to a function, the caller must pass them in the correct order to ensure that the function can process the values as expected.

In the example below, the function `format_slogan` is called but the arguments are passed to it in the wrong order. The resulting computation returns an unexpected and incorrect value.

```
cereal = 'Lucky Charms'
slogan = 'They’re always after me lucky charms' # General Mills Lucky
Charms leprechaun
lucky_charms = format_slogan(slogan, cereal) # Oops! string reversed
```

1.5 Keyword arguments

The caller can pass *keyword arguments*, specifying both a key and value in the form `key=value`. The argument is denoted by its keyword rather than by its position in the argument list.

Keyword arguments free you from having to worry about correctly ordering your arguments in the function call, and they clarify the role of each value in the function call.

❗ note that by convention keyword arguments do not include spaces on either side of the assignment operator

```
lucky_charms = format_slogan(slogan=slogan, name=cereal) # pass args by
keyword

print(f"\n1.5 {lucky_charms}")
```

❗ Take heed, the auto grader will check your keyword argument styling occasionally. Style your keyword arguments as follows:

`some_var = some_func(keyword_arg_01=value_01, keyword_arg_02=value_02)`

not

`some_var = some_func(keyword_arg_01 = value_01, keyword_arg_02 = value_02)`

1.6 Optional parameters

As noted above, a function definition can specify one or more parameter values. Each paramter can specify a default value. In such cases, the caller is not required to pass in a corresponding argument unless a need exists to override the default value.

In the example below, the `format_slogan` parameter named `separator` is provisioned with a default value (`': '`). The caller need only the cereal name and slogan to the function if the `separator` parameter's default value meets their computational requirements.

❗ optional parameters should be listed *after* required parameters in order to allow required arguments to be passed solely by position.

```
def format_slogan(name, slogan, separator=': '):
    return f"{name}{separator}{slogan}"

cereal = 'Honey-nut Cheerios'
slogan = 'Bee Happy, Bee Healthy'

honey_nut_cheerios = format_slogan(cereal, slogan) # use default separator
honey_nut_cheerios = format_slogan(cereal, slogan, '\n') # newline escape
character passed
```

❗ If a function definition includes *multiple* optional parameters, keyword arguments *must* be employed whenever a preceeding optional argument is skipped and not passed to the function by the caller.

```
def format_slogan(name, slogan, separator=': ', all_caps=False):
    if all_caps:
        return f"{name.upper()}{separator}{slogan.upper()}"
    else:
        return f"{name}{separator}{slogan}"

cereal = 'Cinnamon Toast Crunch'
slogan = 'Unlock the Cinnaverse'

# WARN: < all_caps > keyword argument is required since < separator > is
not specified
cinn_toast_crunch = format_slogan(cereal, slogan, all_caps=True)
```

Challenge 01

Task. Write a function that returns cereal brand names by the name of their manufacturer.

1. Implement a function named `get_cereals_by_company` that defines two parameters:

   - `cereal_brands` (`list`): a list of cereal brands
   - `company` (`str`): the name of a cereal producer

   The function returns a list of zero, one, or more cereal brands (name *only*) produced by the passed in company.

2. Implement the accumulator pattern inside the function block in order to iterate over the passed in `cereal_brands`, filter on the `company` name, and accumulate the company's cereal brand names in a "local" list assigned to a variable name of your own choosing.

3. ❗ The `if` statement's expression *must* be able to accommodate passed in company names like "Post", or "post", or "Post Consumer Brands" or "Quaker", "quaker oats", or "Quaker Oats Company".

4. After the loop terminates add a `return` statement that returns the list of cereal brand names.

5. Call the function `get_cereals_by_company` and retrive all cereal brands in `cereals` produced by Post Consumer Brands. Assign the function's return value to the variable named `post_cereals`.

6. Call the function a second time and retrive all cereal brands in `cereals` produced by the Kellogg Company. Assign the function's return value to the variable named `kellogg_cereals`.

7. Uncomment the accompanying `print()` function calls and check your work.

```
cereals = [
    ['General Mills', 'Cocoa Puffs'],
    ['Kellogg Company', 'Frosted Flakes'],
    ['Post Consumer Brands', 'Grape-nuts'],
    ['Kellogg Company', 'Raisin Bran'],
    ['General Mills', 'Cheerios'],
    ['Post Consumer Brands', 'Shredded Wheat (spoon size)'],
    ['General Mills', 'Lucky Charms'],
    ['Quaker Oats Company', "Cap'n Crunch"]
    ]

# TODO Implement function

post_cereals = None # TODO Call function
kellogg_cereals = None # TODO Call function
```

2.0 Calling a function inside a loop

A `for` loop code block can include statements that call functions. For example, if you wanted to return a string for each cereal and its ingredients:

`< Cereal name > ingredients: <Ingredient 01, Ingrediant 02, Ingredient 03 . . . >`

Example: 'Raisin Bran ingredients: Whole Grain Wheat, Raisins, Wheat Bran, Sugar, High Fructose Corn Syrup'

You can assign the task of formatting the ingredients as a string to a function and then call the function from inside a `for` loop for every cereal encountered in `cereals`. Pass a `cereal` list to it and the function returns a formatted string.

```
cereals = [
    ['company_name', 'product_name', 'ingredients', 'serving_size_gm',
'calories', 'sodium_mg', 'sugar_gm', 'protein_gm'],
    ['Kellogg Company', 'Frosted Flakes', ['Milled Corn', 'Sugar', 'Malt
Flavoring', 'High Fructose Corn Syrup', 'Salt'], 37, 130, 190, 12, 2],
    ['Kellogg Company', 'Raisin Bran', ['Whole Grain Wheat', 'Raisins',
'Wheat Bran', 'Sugar', 'High Fructose Corn Syrup'], 59, 190, 200, 17, 5],
    ['General Mills', 'Cheerios', ['Whole Grain Oats', 'Modified Corn
Starch', 'Sugar', 'Salt'], 36, 140, 230, 12, 3],
    ['General Mills', 'Cocoa Puffs', ['Whole Grain Corn', 'Sugar', 'Corn
Syrup', 'Cornmeal', 'Canola and or Rice Bran Oil'], 36, 140, 130, 12, 2],
    ['General Mills', 'Lucky Charms', ['Oats', 'Marshmallows', 'Sugar',
'Corn Syrup', 'Corn Starch'], 36, 140, 230, 12, 3],
    ['Post Consumer Brands', 'Shredded Wheat (spoon size)', ['Whole Grain
Wheat'], 60, 210, 0, 0, 7],
    ['Post Consumer Brands', 'Grape-nuts', ['Whole Grain Wheat', 'Flour',
'Malted Barley Flour', 'Salt', 'Dried Yeast'], 58, 200, 280, 5, 6]
```

```
    ]

def format_ingredients(cereal):
    return ', '.join(cereal[2])

cereal_ingredients = []
for cereal in cereals[1:]:
    cereal_name = cereal[1]
    ingredients = format_ingredients(cereal) # call function
    cereal_ingredients.append(f"{cereal_name} ingredients: {ingredients}")
```

The advantage of delegating the task to a function is that you can call the function from other places in your code, eliminating the need to call `str.join()` every time there is a need to format the list of ingredients as a string. You simply call the function named `format_ingredients` and pass it a `cereal` object whenever you need to build the ingredients string.

3.0 Functions calling other functions

Ideally, a function should perform a single task (i.e., computation), delegating all related tasks to other functions to perform.

For example, if you want to know which cereals are made with corn syrup, you can write a function to perform the task. In this case implement a function that iterates over a cereal's list of ingredients and checks if one of the listed ingredients *contains* the words "Corn Syrup". If Corn Syrup is detected the function returns `True`, otherwise it returns `False`.

💡 for this example the function named `has_corn_syrup` will perform a *case-insensitive* string comparison.

You can write a second function named `get_cereals_with_corn_syrup` that accepts a list of cereals. This function needs to iterate over the passed in list, and check whether or not each cereal element contains corn syrup. Rather than perform the task of iterating over each cereal's list of ingredients, it delegates that task to the function `has_corn_syrup`. Since `has_corn_syrup` returns either `True` or `False` the function call (an expression) can be embedded directly in the `if` statement.

💡 Given that a function call is an *expression*, functions that return either `True` or `False` can be included in an `if` statement.

```
def has_corn_syrup(cereal):
    has_corn_syrup = False

    for ingredient in cereal[2]:
        if 'corn syrup' in ingredient.lower():
            has_corn_syrup = True
            break

    return has_corn_syrup
```

```
def get_cereals_with_corn_syrup(cereals):
    results = []

    for cereal in cereals:
        if has_corn_syrup(cereal): # delegates question to another
function
            results.append(cereal)

    return results

# call function
cereals_with_corn_syrup = get_cereals_with_corn_syrup(cereals)
```

Challenge 02

Task: write two new functions that allow the caller to return a list of cereals that contain a a specified ingredient.

1. Implement a "helper" function named `has_ingredient` that defines two parameters:
   - `ingredients` (`list`): a list of cereal ingredients
   - `ingredient` (`str`): a specific ingredient
2. The function *must* perform a case-insensitive string comparison between the passed in `ingredient` and all elements in `ingredients`. Partial matches are permitted (e.g., `Corn Syrup` will match against `High Fructose Corn_Syrup`).
3. As soon as a match is obtained, exit the loop in order to conserve system resources.
4. The function returns either `True` (has ingredient) or `False` (lacks ingredient).
5. Implement a function named `get_cereals_by_ingredient` that defines two parameters:
   - `cereals` (`list`): representation of a cereal product
   - `ingredient` (`str`): represents a cereal ingredient
6. The function *must* check the ingredients of every cereal in `cereals`. It will delegate the task of checking each cereal's list of ingredients to the function named `has_ingredient`. If `has_ingredient` returns `True`, append the cereal to a "local" accumulator list defined in the function block.
7. After each cereal in `cereals` is evaluated, return the new list of cereals made with the specified ingredient. If no cereals are matched, return an empty list.
8. After implementing the functions, call `get_cereals_by_ingredient` and pass to it the `cereals` list and the string `oats` as arguments. Assign the return value to the variable `cereals_with_oats`.

```
# TODO Implement helper function

# TODO Implement function
```

```
cereals_with_oats = None # TODO Call function
```

Challenge 03

Task: write a "helper" function that allows the caller to return a cereal attribute (e.g., name, ingredients, sugar content, etc.) based on a "header" value (e.g., `product_name`, `sodium_mg`, etc.). Then loop over the `cereals` list and return a list of *one or more* cereals that contain the highest sugar content measured in grams (gm). Ties, if any, are permitted.

1. Extract the headers from `cereals` and assign the list to a variable named `headers`.

2. Implement a function named `get_cereal_attribute` that defines three parameters:

    - `cereal` (`list`): representation of a cereal product
    - `headers` (`list`): represents column headers used to identify cereal attribute values
    - `header` (`str`): a `headers` list element value

    The function will return a `cereal` attribute by its index position in the list, employing the the `header` argument to "look up" the index value.

    💡 the function block can be expressed in one line of code.

3. Create two "local" variables in the function block: `max_sugar` (empty `list`) and `max_sugar_gm` (value = 0). These two "accumulator" variables will help identify the cereal(s) with the highest sugar content measured in grams (gm).

4. Implement a `for` loop that iterates *over the cereals* in the `cereals` list. Write `if-elif` or `if-else-elif` statements that evaluate each cereal encountered according to the following conditions:

5. Access each cereal's `sugar_gm` value by calling the function `get_cereal_attribute` and passing to it the required arguments.

6. if the current cereal's sugar content is *greater* than the previous cereal's sugar content then:

    - *clear* the `max_sugar` list of all existing elements
    - append the cereal's *name* to `max_sugar`
    - update `max_sugar_grm` with the cereal's `sugar_gm` value (`int`)

    💡 there is a handy `list` method that you can call to remove all elements from a list.

7. if the current cereal's sugar content is *equal* to the previous cereal's sugar content then:

    - append the cereal's *name* to `max_sugar`

8. Otherwise, continue on to the next iteration of the loop or exit the loop if the last cereal in `cereals` has been evaluated.

9. Uncomment the accompanying `print()` function and print the cereal(s) with the highest sugar content to the terminal screen.

```
headers = None # TODO Extract headers

# TODO Implement function

cereals_max_sugar = []
max_sugar_grm = 0

# TODO Implement loop
```