


CSC110 Lecture 4: Defining Functions

 Print this handout

Exercise 1: Reviewing the parts of a function definition

You are given the following function definition of a Python function.

```
def calculate(x: int, y: int) -> list:
    """Return a list containing the sum and product of the two given numbers.
    """
    return [x + y, x * y]
```

Answer the following questions about this definition.

1. What is the *function name* in this definition?
2. What is the *function header* in this definition?
3. How many *parameters* does this function have? What is the *name* and *type* of each parameter?
4. What is the function's *return type*?
5. What is the part surrounded by triple-quotes (""") called? What is its purpose?
6. What is the *function body* in this definition?
7. Compared to the example function definitions we looked at in lecture, what part is missing from this function definition?
8. Write down what you would add to complete this function definition.

Exercise 2: The Function Design Recipe

For your reference, here are the steps of the *Function Design Recipe*:

1. Write examples uses
2. Write the function header
3. Write the function description
4. Implement the function body
5. Test the function

1. Consider the following problem description:

Given two lists, return whether they have the same length.

We have started the Function Design Recipe for you. Complete the code below to solve this problem by following the steps of the Function Design Recipe.

```
def is_same_length(list1: list, list2: list) -> bool:
    """
    """

    >>> is_same_length([1, 2, 3], [4, 5, 6])
    True

    >>> is_same_length([1, 2, 3], [])
    False
    """
```

2. Now consider this problem description:

Given a **float** representing the price of a product and another **float** representing a tax rate (e.g., a 10% tax rate represented as the **float** value **0.1**), calculate the after-tax cost of the product, rounded to two decimal places.

Again, complete the code below to solve this problem by following the steps of the Function Design Recipe.

```
def after_tax_cost(price: float, tax_rate: float) -> float:
    """
    """

    >>> after_tax_cost(5.0, 0.01)
    5.05
    """
```

3. Finally, consider this problem description:

Given a list of product prices (as **floats**) and a tax rate, calculate the *total* after-tax cost of all the products in the list rounded to two decimal places.

To keep things simple, don't worry about any rounding errors in your calculation.

This time, we haven't provided you any code—follow the Function Design Recipe to write a complete function that solves this problem.

Tip: to keep your doctest examples numerically simple, try using a tax rate of 0.5 or 1.0.

Exercise 3: Practice with methods

Note: you might want to use [Appendix A.2 Python Built-In Data Types Reference](#) as a reference in this exercise.

1. Suppose we have executed the following assignment statements in the Python console:

```
>>> wish = 'Happy Birthday'
>>> set1 = {1, 2, 3}
>>> set2 = {2, 4, 5, 10}
>>> strings = ['cat', 'rabbit', 'dog', 'rabbit']
```

Write down what each of the following expressions evaluate to.

Do this by hand first! (Then check your work in the Python console. Don't worry if you wrote set elements in a different order that what appears in the Python console.)

```
>>> str.lower(wish)

>>> str.lower(wish[0]) + str.lower(wish[6])

>>> set.union(set1, set2)

>>> set.intersection(set1, set2)

>>> list.count(strings, 'rabbit')
```

2. Suppose we have defined the following variable in the Python console:

```
>>> strings = ['David', 'Tom', 'cool']
```

Write a comprehension that produces a new list with the same elements as **strings**, except with each string converted to uppercase, using the **str.upper** method.

```
>>>

['DAVID', 'TOM', 'COOL']
```

3. Use the Function Design Recipe to solve the following problem:

Given a list of strings, return a new list with the same strings as the input list, except with each string converted to uppercase.

Additional exercises

1. For each of the function definitions given below, complete the definition by writing a description and one doctest example.

```
def multiply_sums(set1: set, set2: set) -> int:
    """
    """

    return sum(set1) * sum(set2)

def exponentiate(nums: list) -> list:
    """
    """

    return [x ** x for x in nums]
```

2. Complete each of the following functions according to their docstring descriptions (this includes the doctests).

```
def different_sums(set1: set, set2: set) -> bool:
    """Return whether set1 and set2 have different sums.

    >>> different_sums({1, 2, 3}, {5, -1})
    True
    >>> different_sums({3}, {1, 2})
    False
    """

def squares(n: int) -> dict:
    """Return a dictionary mapping the numbers from 1 to n to their squares.

    Assume that n > 1.

    >>> squares(3)
    {1: 1, 2: 4, 3: 9}
    """
```

3. Consider the code snippets below. For each code snippet, complete the docstring and/or docstring examples. Use the type contracts, function name, parameter names, and function body to help you.

```
def check_lengths(strings: list, max_length: int) -> bool:
    """
    """

    >>> check_lengths(['cat', 'no', 'maybe'], 4)

    >>>

    """
    return max(len(x) for x in strings) < max_length

def string_lengths(strings: list) -> dict:
    """
    """

    >>> string_lengths(['aaa', 'david'])

    >>>

    """
    return {s: len(s) for s in strings}
```

4. Using the Function Design Recipe, write a function that solves the given problem:

Given a set of strings, calculate the length of the longest string.

5. Using the Function Design Recipe, write a function that solves the given problem:

Given a dictionary mapping names of products (as strings) to prices (as **floats**), which represents a customer order at a store, and a tax rate, calculate the *total* after-tax cost of the products.

Hint: the following illustrates how to use a dictionary as the collection in a comprehension.

```
>>> mapping = {'a': 1, 'b': 2, 'c': 3}
>>> [k for k in mapping] # Variable k is assigned to each key in mapping
['a', 'b', 'c']
>>> [mapping[k] for k in mapping]
[1, 2, 3]
```

6. Suppose you have a variable **greeting** that refers to a string consisting of a mix of letters, numbers, and punctuation (e.g., 'H3ll0!') Write a comprehension that evaluates to a **list** of **bools** that indicates whether each character in the string is alphanumeric, in the same order as they appear in the string.