CSC110 Lecture 7: Function Specification and Working with Definitions

Hisbaan Noorani

September 28, 2020

Contents

1 Ex 1: Reviewing functions correctenss and writing preconditions

1. What is the relationship between a function's paramater type annotations and a function's preconditions?

   The paramater type annotations are one of the functions preconditions. The paramaters must have the specified type.

2. The following function calculates the pay for an employee who worked for a given time period (e.g., 10am–4pm) at a given hourly pay rate (e.g., $15/hour). Write Python expressions for preconditions to express the constraints on the function inputs described in the docstring.

```python
def calculate_pay(start: int, end: int, pay_rate: float) -> float:
    """Return the pay of an employee who worked for the given time at the given pay rate.

    start and end represent the hour (from 0 to 23 inclusive) that the employee
    started and ended their work.

    pay_rate is the hourly pay rate, and must be >= 15.0 (the minimum wage).

    Preconditions:
      - pay_rate >= 15.0
      - 0 <= start <= 23
      - 0 <= end <= 23
      - start <= end

    >>> calculate_pay(3, 5, 15.5)
    31.0
    >>> calculate_pay(9, 21, 22.0)
    264.0
    """
    return (end - start) * pay
```

3. Implement the function below.

```
def ticket_price(age: int) -> float:
    """Return the ticket price for a person who is age years old.

    Seniors 65 and over pay 4.75, kids 12 and under pay 4.25, and
    everyone else pays 7.50.

    Precondition:
      - age > 0

    >>> ticket_price(7)
    4.25
    >>> ticket_price(21)
    7.5
    >>> ticket_price(101)
    4.75
    """
    elif age <= 12:
        return 4.25
    elif age >= 65:
        return 4.75
    else:
        return 7.50
```

2 Ex 2: `typing` type annotations

1. For each of the following python literal values, write down the approrpriate type annotation (from `typing`) for that value.

Python Value	Type annotation
`[1, 2, 3]`	List[int]
`{'hi', 'bye', 'haha'}`	Set[str]
`{1.5: True, 3.6: False, -1.0: True}`	Dict[float, str]
`(1, 'Hi')`	Tuple[int, str]
`([1, 2, 3], [4, 5, 6])`	Tuple[List[int], List[int]]

2. For each of the following pieces of (collections) data, write the appropriate type annotation using the `typing` module to represent that data.

Description of Data	Type Annotation
A study music playlist (song names)	Set[str]
A colour in the RGB24 model	Tuple[int, int, int]
David's grocery list (food names and quantities)	Dict[str, int]
An unordered collection of distinct points in the Cartesian plane	Set[Tuple[int, int]]

3. Why would we type the annotation `list` instead of `List[...]` (with a type in the square brackets)?

   A list may contain multiple element types so we can't descirbe all of them with `List[...]`

3 Ex 3:

1. Consider the following statement:

If m and n are odd integers, then mn is an odd integer.

If we want to express this statement using predicate logic, we need to start with a definition of the term "odd". Let $n \in \mathbb{Z}$. We say that n is odd when $2|(n-1)$. That is, n is odd when $\exists k \in \mathbb{Z}, n = 2k+1$

(a) Write the definiton of a predicate over the integers named *Odd* that is true when its argument is odd.
$Odd : \exists k \in \mathbb{Z}, n = 2k+1$, where $n \in \mathbb{Z}$
or
$Odd : 2|(n-1)$, where $n \in \mathbb{Z}$

(b) Using the predicate *Odd* and the notation of predicate logic, express the statement:

For every pair of odd integers, m and n, mn is an odd integer.

$\forall m, n \in \mathbb{Z}, (Odd(m) \wedge Odd(n)) \Rightarrow Odd(mn)$

(c) Repeat part (b) but do not use the predicates *Odd* or $|$. Instead use the full definition of odd that includes a quantified variable.
$\forall m, n \in \mathbb{Z}, \exists k_1, k_2, k_3 \in \mathbb{Z}$ s.t. $(m = 2k_1 + 1 \wedge n = 2k_2 + 1) \Rightarrow mn = 2k + 1$
or
$\forall m, n \in \mathbb{Z}, ((\exists k_1 \in \mathbb{Z} \text{ s.t. } m = 2k_1 + 1) \wedge (\exists k_2 \in \mathbb{Z} \text{ s.t. } n = 2k_2 + 1)) \Rightarrow (\exists k_3 \in \mathbb{Z} \text{ s.t. } mn = 2k_3 + 1)$

2. consider the following Python functions.

```python
def average(nums: Set[int]) -> float:
    """Return the average of a set of numbers. (Preconditions omitted)"""
    return sum(nums) / len(nums)


def larger_average(nums1: Set[int], nums2: Set[int]) -> Set[int]:
    """Return the set of numbers with the larger average. (Preconditions omitted)"""
    if average(nums1) >= average(nums2):
        return nums1
    else:
        return nums2
```

Rewrite `larger_average` so that it does not call `average`, but instead does the same calculation as the body of average directly.

(This is the equivalent of "expanding" the definition of a function.)

```python
def larger_average(nums1: Set[int], nums2: Set[int]) -> Set[int]:
    """Return the set of numbers with the larger average. (Preconditions ommitted)"""
    if sum(nums1) / len(nums1) >= sum(nums2) / len(nums2):
        return nums1
    else:
        return nums2
```

4 Additional Exercises

1. Repeat Exercise 3 questions 3b and 3c using the following statement (which states the converse of the original implication from that question).

   For every pair of integers m and n, mn is odd, then m and n are odd.

2. Now consider a similar computational task.

(a) Define a Python function that takes an `int` and erturns whether it is odd.

(b) Define a Python function that takes a set of ~int~s, and returns a new set containing the elements of the input set that are odd.

Implement this function in two ways: using the function you defined in part (a), and not using that funciton (instead writing its body in your function for this part).