Homework 2: Higher Order Functions

hw02.zip (hw02.zip)

Due by 11:59pm on Thursday, September 7

Instructions

Download hw02.zip (hw02.zip). Inside the archive, you will find a file called hw02.py (hw02.py), along with a copy of the `ok` autograder.

Submission: When you are done, submit the assignment by uploading all code files you've edited to Gradescope. You may submit more than once before the deadline; only the final submission will be scored. Check that you have successfully submitted your code on Gradescope. See Lab 0 (/lab/lab00#submitting-the-assignment) for more instructions on submitting assignments.

Using Ok: If you have any questions about using Ok, please refer to this guide. (/articles/using-ok)

Readings: You might find the following references useful:

- Section 1.6 (https://www.composingprograms.com/pages/16-higher-order-functions.html)

Grading: Homework is graded based on correctness. Each incorrect problem will decrease the total score by one point. There is a homework recovery policy as stated in the syllabus. **This homework is out of 2 points.**

Required Questions

Getting Started Videos

Several doctests refer to these functions:

```
from operator import add, mul

square = lambda x: x * x

identity = lambda x: x

triple = lambda x: 3 * x

increment = lambda x: x + 1
```

Higher Order Functions

Q1: Product

Write a function called `product` that returns the product of the first `n` terms of a sequence. Specifically, `product` takes in an integer `n` and `term`, a single-argument function that determines a sequence. (That is, `term(i)` gives the `i` th term of the sequence.) `product(n, term)` should return `term(1) * ... * term(n)`.

```
def product(n, term):
    """Return the product of the first n terms in a sequence.

    n: a positive integer
    term:  a function that takes one argument to produce the term

    >>> product(3, identity)  # 1 * 2 * 3
    6
    >>> product(5, identity)  # 1 * 2 * 3 * 4 * 5
    120
    >>> product(3, square)    # 1^2 * 2^2 * 3^2
    36
    >>> product(5, square)    # 1^2 * 2^2 * 3^2 * 4^2 * 5^2
    14400
    >>> product(3, increment) # (1+1) * (2+1) * (3+1)
    24
    >>> product(3, triple)    # 1*3 * 2*3 * 3*3
    162
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q product
```

Q2: Accumulate

Let's take a look at how `product` is an instance of a more general function called `accumulate`, which we would like to implement:

```
def accumulate(merger, start, n, term):
    """Return the result of merging the first n terms in a sequence and start.
    The terms to be merged are term(1), term(2), ..., term(n). merger is a
    two-argument commutative function.

    >>> accumulate(add, 0, 5, identity)  # 0 + 1 + 2 + 3 + 4 + 5
    15
    >>> accumulate(add, 11, 5, identity) # 11 + 1 + 2 + 3 + 4 + 5
    26
    >>> accumulate(add, 11, 0, identity) # 11
    11
    >>> accumulate(add, 11, 3, square)   # 11 + 1^2 + 2^2 + 3^2
    25
    >>> accumulate(mul, 2, 3, square)    # 2 * 1^2 * 2^2 * 3^2
    72
    >>> # 2 + (1^2 + 1) + (2^2 + 1) + (3^2 + 1)
    >>> accumulate(lambda x, y: x + y + 1, 2, 3, square)
    19
    >>> # ((2 * 1^2 * 2) * 2^2 * 2) * 3^2 * 2
    >>> accumulate(lambda x, y: 2 * x * y, 2, 3, square)
    576
    >>> accumulate(lambda x, y: (x + y) % 17, 19, 20, square)
    16
    """
    "*** YOUR CODE HERE ***"
```

`accumulate` has the following parameters:

- `merger`: a two-argument function that specifies how the current term is merged with the previously accumulated terms
- `start`: value at which to start the accumulation
- `n`: an integer indicating the number of terms to merge
- `term`: a single-argument function that determines a sequence; `term(i)` is the `i` th term of the sequence

`accumulate` should merge the first `n` terms of the sequence defined by `term` with the `start` value according to the `merger` function.

For example, the result of `accumulate(add, 11, 3, square)` is

```
11 + square(1) + square(2) + square(3) = 25
```

> **Note:** You may assume that `merger` is commutative. That is, `merger(a, b) == merger(b, a)` for all `a` and `b`. However, you may not assume `merger` is chosen from a fixed function set and hard-code the solution.

After implementing `accumulate`, show how `summation` and `product` can both be defined as function calls to `accumulate`.

Important: You should have a single line of code (which should be a `return` statement) in each of your implementations for `summation_using_accumulate` and `product_using_accumulate`, which the syntax check will check for. You **must** delete the `***YOUR CODE HERE***` placeholder under each function.

```
def summation_using_accumulate(n, term):
    """Returns the sum: term(1) + ... + term(n), using accumulate.

    >>> summation_using_accumulate(5, square)
    55
    >>> summation_using_accumulate(5, triple)
    45
    >>> # You aren't expected to understand the code of this test.
    >>> # Check that the bodies of the functions are just return statements.
    >>> # If this errors, make sure you have removed the "***YOUR CODE HERE***".
    >>> import inspect, ast
    >>> [type(x).__name__ for x in ast.parse(inspect.getsource(summation_using_accumulate)
    ['Expr', 'Return']
    """
    "*** YOUR CODE HERE ***"

def product_using_accumulate(n, term):
    """Returns the product: term(1) * ... * term(n), using accumulate.

    >>> product_using_accumulate(4, square)
    576
    >>> product_using_accumulate(6, triple)
    524880
    >>> # You aren't expected to understand the code of this test.
    >>> # Check that the bodies of the functions are just return statements.
    >>> # If this errors, make sure you have removed the "***YOUR CODE HERE***".
    >>> import inspect, ast
    >>> [type(x).__name__ for x in ast.parse(inspect.getsource(product_using_accumulate))
    ['Expr', 'Return']
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q accumulate
python3 ok -q summation_using_accumulate
python3 ok -q product_using_accumulate
```

Takeaway: Notice how quick it is now to create accumulator functions with different `merger` functions! This is because we abstracted away the logic of `product` and `summation` into the `accumulate` function. Without this abstraction, our code for a `summation` function would be

just as long as our code for the `product` function from Question 1, and the logic would be highly redundant!

Q3: Funception

Implement `funception`, which takes in a function `func1` and a number `begin` and returns a function `func2`. `func2` should take a single argument, `end`, and apply `func1` on all the numbers from `begin` (inclusive) up to `end` (exclusive) and return the *product*.

If `begin` is greater than or equal to `end`, the range of numbers is invalid, and `func2` should return `1`.

> **Note 1:** While similar to `accumulate`, the function returned by `funception` merges terms over the range `[begin, end)` (rather than `[1, n]` alongside some arbitrary `begin` term).

> **Note 2:** If you attempt to **modify** `begin` in `func2`, you will get an `UnboundLocalError`. You're not allowed to rebind variables defined outside of our current frame! However, you can still **access** the value of `begin` in `func2` and set it equal to a new variable. An example is shown below:

Unbound Local Error:

```
>>> def f():
...     x = 5
...     def g():
...         x += 1 # UnboundLocalError: cannot modify variable X outside current frame
...         print(x)
...     return g
...
>>> h = f()
>>> h()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in g
UnboundLocalError: local variable 'x' referenced before assignment
```

No Error:

```
>>> def f():
...     x = 5
...     def g():
...         i = x # No Error: accesses value X from parent frame, binds it to I in the cu
...         i += 1
...         print(i)
...     return g
...
>>> h = f()
>>> h()
6
```

```
def funception(func1, begin):
    """ Takes in a function (func1) and a begin value.
    Returns a function (func2) that will find the product of
    func1 applied to the range of numbers from
    begin (inclusive) to end (exclusive)

    >>> def increment(num):
    ...     return num + 1
    >>> def double(num):
    ...     return num * 2
    >>> g1 = funception(increment, 0)
    >>> g1(3)    # increment(0) * increment(1) * increment(2) = 1 * 2 * 3 = 6
    6
    >>> g1(0)    # Returns 1 because begin >= end
    1
    >>> g1(-1)   # Returns 1 because begin >= end
    1
    >>> g2 = funception(double, 1)
    >>> g2(3)    # double(1) * double(2) = 2 * 4 = 8
    8
    >>> g2(4)    # double(1) * double(2) * double(3) = 2 * 4 * 6 = 48
    48
    >>> g3 = funception(increment, -3)
    >>> g3(-1)   # increment(-3) * increment(-2) = -2 * -1 = 2
    2
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q funception
```

Lambda Expressions

Q4: Lambda Math

To get familiar with writing `lambda` expressions, let's see how we can use them to help us do some math.

Important: For each of the following problems, your solution should have only a single line of code (which should be a `return` statement). The syntax check will check for this.

We can start by using a `lambda` to complete the `mul_by_num` function.

```
def mul_by_num(num):
    """Returns a function that takes one argument and returns num
    times that argument.

    >>> x = mul_by_num(5)
    >>> y = mul_by_num(2)
    >>> x(3)
    15
    >>> y(-4)
    -8
    """
    return ______
```

`mul_by_num` takes in a single number as an argument, and returns a one argument **function** that multiplies any value passed to it by the original number.

Use Ok to test your code:

```
python3 ok -q mul_by_num
```

The next thing we want to do is write a function that takes in two functions, `f1` and `f2`, and returns another **function** that takes in a single argument `x`. The returned function should compute `f1(x) + f2(x)`. You can assume both `f1` and `f2` take in one argument, and their result can be added together.

```
def add_results(f1, f2):
    """
    Return a function that takes in a single variable x, and returns
    f1(x) + f2(x). You can assume the result of f1(x) and f2(x) can be
    added together, and they both take in one argument.

    >>> identity = lambda x: x
    >>> square = lambda x: x**2
    >>> a1 = add_results(identity, square) # x + x^2
    >>> a1(4)
    20
    >>> a2 = add_results(a1, identity)     # (x + x^2) + x
    >>> a2(4)
    24
    >>> a2(5)
    35
    >>> a3 = add_results(a1, a2)           # (x + x^2) + (x + x^2 + x)
    >>> a3(4)
    44
    """
    return ______
```

Use Ok to test your code:

```
python3 ok -q add_results
```

Finally, let's use a `lambda` expression to create a function that takes in two integers, `x` and `y`, and returns whether or not `x` is evenly divisible by `y`. Complete the function `mod_maker`, which has no input but will return a **function** that, when called on two integers, will return True if `x` is divisible by `y`. Otherwise, it should return the remainder of `x % y`. You may **not** use an `if` or `...if...else...` statement in your lambda expression.

```
def mod_maker():
    """Return a two-argument function that performs the modulo operation and
    returns True if the numbers are divisble, and the remainder otherwise.

    >>> mod = mod_maker()
    >>> mod(7, 2) # 7 % 2
    1
    >>> mod(4, 8) # 4 % 8
    4
    >>> mod(8,4) # 8 % 4
    True
    >>> from construct_check import check
    >>> check(HW_SOURCE_FILE, 'mod_maker', ['If', 'IfExp']) # no if / if-else statements
    True
    """
    return ______
```

> **Note:** You are allowed (and expected) to use the modulo operator itself in your solution. The goal of this function is not to recreate the operator from scratch, but to create an alternate way of calling it.
>
> *Hint:* Recall that Python implements short-circuiting. Additionally, recall that Python values like `False`, `0`, `None`, and `""` are considered false values. Can we utilize an `and` or `or` statement in our `lambda` to solve this problem?

Use Ok to test your code:

```
python3 ok -q mod_maker
```

Use Ok to run the local syntax checker (which checks that each of your solutions only contains one line):

```
python3 ok -q lambda_math_syntax_check
```

Takeaway: Although lambda functions have more restrictions about what they can contain in their bodies than regular functions, they are still useful, especially for making minor tweaks and additions to pre-existing code that doesn't have quite the right structure for what we are trying to do.

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit

Make sure to submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** For a refresher on how to do this, refer to Lab 00 (https://cs61a.org/lab/lab00/#submit-with-gradescope).

Exam Practice

Homework assignments will also contain prior exam questions for you to try. These questions have no submission component; feel free to attempt them if you'd like some practice!

> Note that exams from Spring 2020, Fall 2020, and Spring 2021 gave students access to an interpreter, so the question format may be different than other years. Regardless, the questions below are good problems to try *without* access to an interpreter.

1. Fall 2019 MT1 Q3: You Again (https://cs61a.org/exam/fa19/mt1/61a-fa19-mt1.pdf#page=4) [Higher Order Functions]
2. Spring 2021 MT1 Q4: Domain on the Range (https://cs61a.org/exam/sp21/mt1/61a-sp21-mt1.pdf#page=14) [Higher Order Functions]
3. Fall 2021 MT1 Q1b: tik (https://cs61a.org/exam/fa21/mt1/61a-fa21-mt1.pdf#page=4) [Functions and Expressions]