Homework 5: Data Abstraction, Mutable Functions & Generators **hw05.zip (hw05.zip)**

Due by 11:59pm on Friday, 3/8

Instructions

Download hw05.zip (hw05.zip). All questions are required.

Submission: When you are done, submit with `python3 ok --submit`. 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 okpy.org (https://okpy.org/). 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.html)

Readings: You might find the following references useful:

- Section 2.4 (http://composingprograms.com/pages/24-mutable-data.html)
- Section 2.5 (http://composingprograms.com/pages/25-object-oriented-programming.html)
- Section 4.2 (http://composingprograms.com/pages/42-implicit-sequences.html)

Grading: Homework is graded based on effort, not correctness. However, there is no partial credit; you must show substantial effort on every problem to receive any points.

Mutable functions

Q1: Counter

Define a function `make_counter` that returns a `counter` function, which takes a string and returns the number of times that the function has been called on that string.

```
def make_counter():
    """Return a counter function.

    >>> c = make_counter()
    >>> c('a')
    1
    >>> c('a')
    2
    >>> c('b')
    1
    >>> c('a')
    3
    >>> c2 = make_counter()
    >>> c2('b')
    1
    >>> c2('b')
    2
    >>> c('b') + c2('b')
    5
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q make_counter
```

Q2: Next Fibonacci

Write a function `make_fib` that returns a function that returns the next Fibonacci number each time it is called. (The Fibonacci sequence begins with 0 and then 1, after which each element is the sum of the preceding two.) Use a `nonlocal` statement!

```
def make_fib():
    """Returns a function that returns the next Fibonacci number
    every time it is called.

    >>> fib = make_fib()
    >>> fib()
    0
    >>> fib()
    1
    >>> fib()
    1
    >>> fib()
    2
    >>> fib()
    3
    >>> fib2 = make_fib()
    >>> fib() + sum([fib2() for _ in range(5)])
    12
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q make_fib
```

Q3: Password Protected Account

In lecture, we saw how to use functions to create mutable objects. Here, for example, is the function `make_withdraw` which produces a function that can withdraw money from an account:

```
def make_withdraw(balance):
    """Return a withdraw function with BALANCE as its starting balance.
    >>> withdraw = make_withdraw(1000)
    >>> withdraw(100)
    900
    >>> withdraw(100)
    800
    >>> withdraw(900)
    'Insufficient funds'
    """
    def withdraw(amount):
        nonlocal balance
        if amount > balance:
           return 'Insufficient funds'
        balance = balance - amount
        return balance
    return withdraw
```

Write a version of the `make_withdraw` function that returns password-protected withdraw functions. That is, `make_withdraw` should take a password argument (a string) in addition to an initial balance. The returned function should take two arguments: an amount to withdraw and a password.

A password-protected `withdraw` function should only process withdrawals that include a password that matches the original. Upon receiving an incorrect password, the function should:

1. Store that incorrect password in a list, and
2. Return the string 'Incorrect password'.

If a withdraw function has been called three times with incorrect passwords `p1`, `p2`, and `p3`, then it is locked. All subsequent calls to the function should return:

```
"Your account is locked. Attempts: [<p1>, <p2>, <p3>]"
```

The incorrect passwords may be the same or different:

```
def make_withdraw(balance, password):
    """Return a password-protected withdraw function.

    >>> w = make_withdraw(100, 'hax0r')
    >>> w(25, 'hax0r')
    75
    >>> error = w(90, 'hax0r')
    >>> error
    'Insufficient funds'
    >>> error = w(25, 'hwat')
    >>> error
    'Incorrect password'
    >>> new_bal = w(25, 'hax0r')
    >>> new_bal
    50
    >>> w(75, 'a')
    'Incorrect password'
    >>> w(10, 'hax0r')
    40
    >>> w(20, 'n00b')
    'Incorrect password'
    >>> w(10, 'hax0r')
    "Your account is locked. Attempts: ['hwat', 'a', 'n00b']"
    >>> w(10, 'l33t')
    "Your account is locked. Attempts: ['hwat', 'a', 'n00b']"
    >>> type(w(10, 'l33t')) == str
    True
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q make_withdraw
```

Q4: Joint Account

Suppose that our banking system requires the ability to make joint accounts. Define a function `make_joint` that takes three arguments.

1. A password-protected `withdraw` function,
2. The password with which that `withdraw` function was defined, and
3. A new password that can also access the original account.

The `make_joint` function returns a `withdraw` function that provides additional access to the original account using *either* the new or old password. Both functions draw from the same balance. Incorrect passwords provided to either function will be stored and cause the functions to be locked after three wrong attempts.

Hint: The solution is short (less than 10 lines) and contains no string literals! The key is to call `withdraw` with the right password and amount, then interpret the result. You may assume that all failed attempts to withdraw will return some string (for incorrect passwords, locked accounts, or insufficient funds), while successful withdrawals will return a number.

Use `type(value) == str` to test if some `value` is a string:

```
def make_joint(withdraw, old_password, new_password):
    """Return a password-protected withdraw function that has joint access to
    the balance of withdraw.

    >>> w = make_withdraw(100, 'hax0r')
    >>> w(25, 'hax0r')
    75
    >>> make_joint(w, 'my', 'secret')
    'Incorrect password'
    >>> j = make_joint(w, 'hax0r', 'secret')
    >>> w(25, 'secret')
    'Incorrect password'
    >>> j(25, 'secret')
    50
    >>> j(25, 'hax0r')
    25
    >>> j(100, 'secret')
    'Insufficient funds'

    >>> j2 = make_joint(j, 'secret', 'code')
    >>> j2(5, 'code')
    20
    >>> j2(5, 'secret')
    15
    >>> j2(5, 'hax0r')
    10

    >>> j2(25, 'password')
    'Incorrect password'
    >>> j2(5, 'secret')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    >>> j(5, 'secret')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    >>> w(5, 'hax0r')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    >>> make_joint(w, 'hax0r', 'hello')
    "Your account is locked. Attempts: ['my', 'secret', 'password']"
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

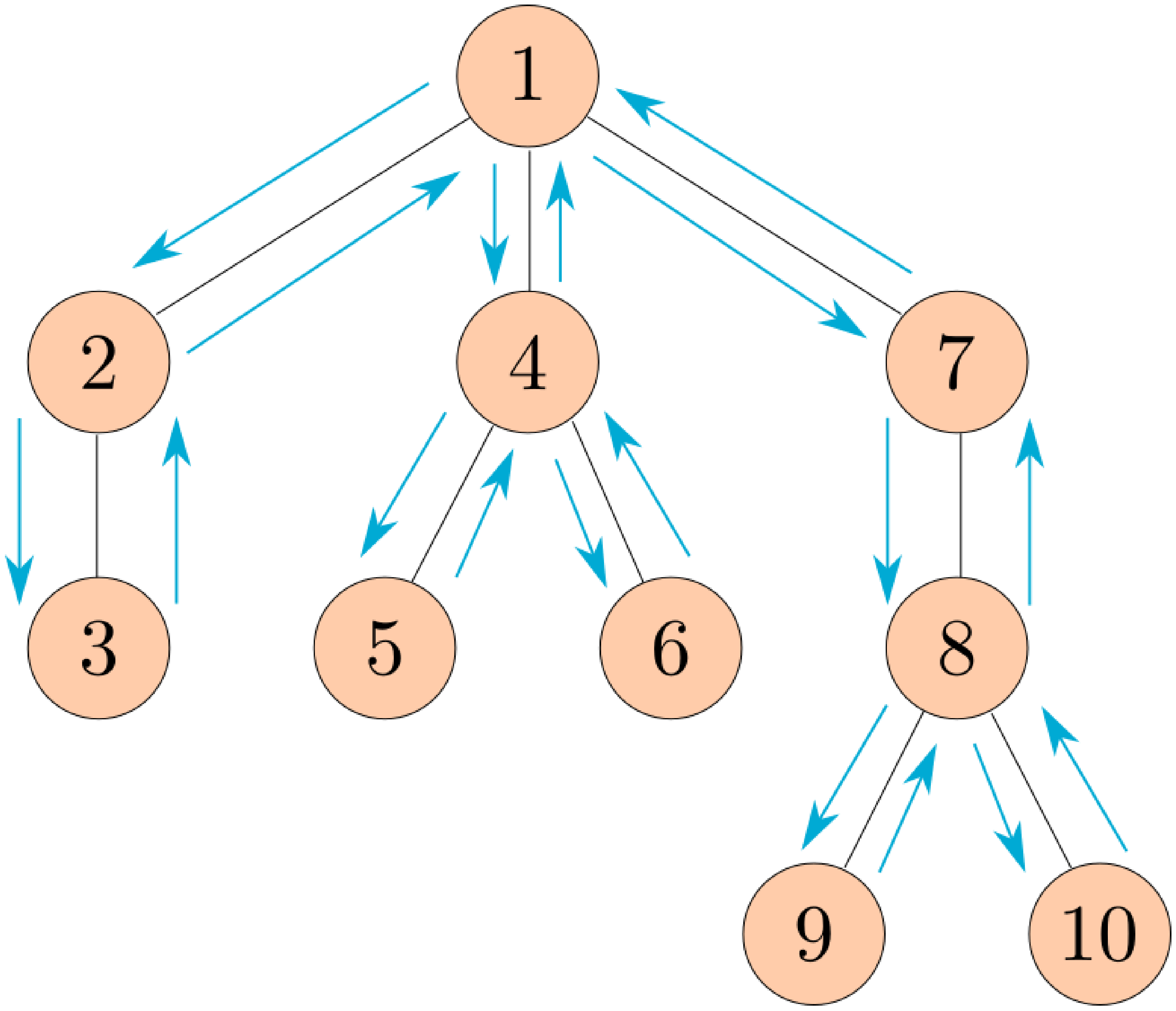
```
python3 ok -q make_joint
```

Trees

Q5: Preorder

Define the function `preorder`, which takes in a tree as an argument and returns a list of all the entries in the tree in the order that `print_tree` would print them.

The following diagram shows the order that the nodes would get printed, with the arrows representing function calls.



> *Note*: This ordering of the nodes in a tree is called a preorder traversal.

```
def preorder(t):
    """Return a list of the entries in this tree in the order that they
    would be visited by a preorder traversal (see problem description).

    >>> numbers = tree(1, [tree(2), tree(3, [tree(4), tree(5)]), tree(6, [tree(7)])])
    >>> preorder(numbers)
    [1, 2, 3, 4, 5, 6, 7]
    >>> preorder(tree(2, [tree(4, [tree(6)])]))
    [2, 4, 6]
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q preorder
```

Objects

Q6: Mint

Complete the `Mint` and `Coin` classes so that the coins created by a mint have the correct year and worth.

- Each `Mint` instance has a `year` stamp. The `update` method sets the `year` stamp to the `current_year` class attribute of the `Mint` class.
- The `create` method takes a subclass of `Coin` and returns an instance of that class stamped with the `mint` 's year (which may be different from `Mint.current_year` if it has not been updated.)
- A `Coin` 's `worth` method returns the `cents` value of the coin plus one extra cent for each year of age beyond 50. A coin's age can be determined by subtracting the coin's year from the `current_year` class attribute of the `Mint` class.

```
class Mint:
    """A mint creates coins by stamping on years.

    The update method sets the mint's stamp to Mint.current_year.

    >>> mint = Mint()
    >>> mint.year
    2017
    >>> dime = mint.create(Dime)
    >>> dime.year
    2017
    >>> Mint.current_year = 2100  # Time passes
    >>> nickel = mint.create(Nickel)
    >>> nickel.year     # The mint has not updated its stamp yet
    2017
    >>> nickel.worth()  # 5 cents + (83 - 50 years)
    38
    >>> mint.update()   # The mint's year is updated to 2100
    >>> Mint.current_year = 2175     # More time passes
    >>> mint.create(Dime).worth()    # 10 cents + (75 - 50 years)
    35
    >>> Mint().create(Dime).worth()  # A new mint has the current year
    10
    >>> dime.worth()     # 10 cents + (160 - 50 years)
    118
    >>> Dime.cents = 20  # Upgrade all dimes!
    >>> dime.worth()     # 20 cents + (160 - 50 years)
    128

    """
    current_year = 2017

    def __init__(self):
        self.update()

    def create(self, kind):
        "*** YOUR CODE HERE ***"

    def update(self):
        "*** YOUR CODE HERE ***"

class Coin:
    def __init__(self, year):
        self.year = year

    def worth(self):
        "*** YOUR CODE HERE ***"

class Nickel(Coin):
    cents = 5

class Dime(Coin):
    cents = 10
```

Use Ok to test your code:

```
python3 ok -q Mint
```

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)