CS 2001 - Python Test

Fall 2016

Name: _____

Instructions[1]:

1. Read all instructions or suffer the consequences.
2. No materials other than your pencil and eraser are allowed or required for this test.
3. Do not cheat. Don't even try.
4. Contain any/all Pokémon in their respective Poké Balls for the duration of the test. Any damage or disturbance caused by escaped creatures will earn a zero for their respective trainer.
5. Place answers in blanks if they are provided. If you are asked to write a short answer, make it literate. I have little patience for illiterumcy-nesses. Correctify all yoor missed steaks.
6. If you have a question about a problem/question, be sure to ask.
7. If you are asked to give code for a problem, give all pertinent code. Do not worry about commenting your code unless asked specifically to do so.
8. Good luck.

Problem 0

Choose the best option for each of the following. Clearly indicate your selection.

1. (5 pts) Python syntax permits an `if`-statement to have an optional `else if` as shown below

```
>>> if a < 5:
>>>     print("whatever. it's true, you are whack.")
>>> else if a > 10:
>>>     print("toot toot here comes books")
```

   a. True
   b. False

2. (5 pts) What is the output of the following?

```
>>> def fun(a):
>>>     a.append("grapes")
>>>     a = []
>>>     a.append("avocados")
>>>
>>> lst = ["walnuts"]
>>> fun(lst)
>>> print(lst)
```

   a. `["walnuts"]`
   b. `["walnuts", "grapes"]`
   c. `["avocados"]`
   d. `["walnuts", "grapes", "avocados"]`
   e. None of the above

[1] Shamelessly nabbed from one "`cp`".

Problem 1

Consider the following code:

```
f = None
try:
    f = open("input.txt")
    print("It OK!")
except KeyError:
    print("Caught a KeyError!")
except ZeroDivisionError:
    print("Caught a ZeroDivisionError!")
except Exception:
    print("Something bad happened!")
```

1. (2 pts) What is output if "input.txt" exists and contains the string "potato"?

2. (3 pts) What is output if "input.txt" doesn't exist?

3. (5 pts) How would you improve the error handling in the code above? Why do your modifications improve the current implementation?

Problem 2

Do the following:

1. Write two snippets of code:
   - Open and write the string "frog" to a file named "data.txt" using a `with`-statement.
   - Then, open the same file, and print its contents *without* using a `with`-statement.

```
# (3 pts) Using a with-statement
```

```
# (3 pts) Without using a with-statement
```

2. (4 pts) Why would one choose to use a `with`-statement with `open()`?

Problem 3

Do the following:

1. (10 pts) In the space indicated below, implement `drop_while` as described.
2. (5 pts) In the blank space provided, write a lambda, so that the line below returns the expected list. The lambda should be the first argument to `drop_while`. `my_list` is defined in the example at the bottom of this page.

```
list(drop_while(                                    , my_list))  # returns [4, 3, 14, 83]
```

```
def drop_while(test, my_iterable):
    """Return a generator that yields items that *follow* the longest sequence of items
    (starting at the beginning of my_iterable) for which ``test(item)`` is True.

    In other words, it drops items (starting at the beginning of my_iterable) as long as
    the test function returns True. It yields the remaining items.

    In other other words, the generator yields the items that
    ``take_while(test, my_iterable)`` doesn't.

    :param function test: A function that takes a single parameter and returns False when
        it's time to start yielding items. That is, start yielding items when you find an
        item that fails the test.

    :param iterable my_iterable: An iterable from which to yield items.

    :return: A generator that yields items at the end of my_iterable that follow the last
        item to pass the test function.

    :rtype: generator
    """
    # space indicated
```

```
###
# Example usage
###

def is_even(x):
    return x % 2 == 0

my_list = [0, 2, 6, 7, 10, 4, 3, 14, 83]
list(take_while(is_even, my_list))  # returns [0, 2, 6]
list(drop_while(is_even, my_list))  # returns [7, 10, 4, 3, 14, 83]
```

Problem 4

Consider the following code:

```
# A dictionary that maps keys of type str to values of type str.
bob = {"frog": "twelve", "toothpaste": "walnut", "seaweed": "dust", "cat": "ladybug"}
```

Do the following:

1. (5 pts) Write a **list comprehension** to create a new list that contains all keys from `bob` that contain the letter "a". Name the new list `bob_keys`.

2. (4 pts) What is the order of the items in `bob_keys`?

3. (5 pts) Write a **dictionary comprehension** that creates a new dictionary that maps each key from `bob` to its corresponding string length. Name the new dictionary `jimmy_pesto`.

```
jimmy_pesto == {"seaweed": 7, "toothpaste": 10, "frog": 4, "cat": 3}  # returns True
```

Problem 5

Consider the following code:

```
import functools

def my_decorator(func):
    return functools.partial(func, 4, 3)

@my_decorator
def my_func(a, b, c):
    return a - (b + c)

# down here
my_func(2)
```

Answer the following questions:

1. (5 pts) When we reach the comment `down here`, how many arguments does `my_func` accept? Explain your answer.

2. (10 pts) What would be returned when we call `my_func(2)`? Explain how you reached your answer.

Problem 6

Below is a Poliwag class. Its implementation is highly advanced and irrelevant. Someone has taken the time to define the `__lt__` method for Poliwags, so that we can compare Poliwags[2] using `<` . Poliwags comprise a *totally ordered set*. In fact, they have strict total ordering. That means (among other mathy things)...

- $a < b$ if and only if **not** $b \leq a$.
  - If **not** $b \leq a$ then $a < b$.
  - If **not** $a < b$ then $b \leq a$.

Implement the missing methods, so that we can also use `>`, `>=`, and `<=` to compare Poliwags. You will need to use `<` (which is given) in at least one of the definitions below.

Assume that, despite the redacted implementation details, the class is syntactically correct.

```
class Poliwag:

    # ... all kinds of methods defined here ...

    def __lt__(self, other):
        # ... more implementation secrets ...
        # The less-than operator already works.
        return is_less_than  # a boolean

    def __gt__(self, other_pwag):  # (4 pts)



    def __le__(self, other_pwag):  # (4 pts)



    def __ge__(self, other_pwag):  # (4 pts)
```

Problem 7

Consider the following code snippet:

```
def is_even(x):
    return x % 2 == 0

numbas = ["10", "22", "9", "56", "73", "925"]
f = filter(is_even, map(int, numbas))
```

1. (5 pts) What would be the output if we were to `print(list(f))`[3]?

2. (9 pts) Write a generator expression that yields the same values as `f`.

[2] Individual values are hidden and it's frustrating to everyone.
[3] `f` is a filter object. If we convert it to a list first, we can actually see its contents.

Bonus

(5 pts, all or nothing) In the space indicated below, write down the full output that is written to standard out (i.e., to the console) when the program is executed as indicated.

```
# m1.py
print("{}: Importing m2".format(__name__))
import m2
print("{}: Done importing m2".format(__name__))


def m1_func():
    print("Holy moly")

if __name__ == '__main__':
    m2.m2_func()

print("Finished {}".format(__name__))
```

```
# m2.py
print("{}: Importing m1".format(__name__))
import m1
print("{}: Done importing m1".format(__name__))


def m2_func():
    print("Great Googly Moogly")

if __name__ == '__main__':
    m1.m1_func()

print("Finished {}".format(__name__))
```

```
$ python3.4 m1.py
# The space indicated
```