Homework 1

This homework covers the following topics from class:

- Course Introduction
    - History, course methodology, compilers/interpreters/linkers
- Essential Python
    - Everything you (never learned but) need to know to write correct python code

Each problem below has a time associated with it. This is the amount of time that we expect a thoroughly-prepared student would take to solve this problem on an exam. We understand, however, that as you learn these concepts, these questions may take more time to solve, and we don't want to overwhelm you with homework. For that reason, only **required** questions need to be completed when you submit this homework. That said, the exams will cover all materials covered (a) in class, (b) from class projects, (c) in the required homework problems, and (d) in the optional homework problems, so at a minimum, please review the optional problems and/or use them as exam prep materials.

For every homework, you must turn in a **PDF file** with your answers via Gradescope. You may include both typed and hand-written solutions, so long as they are legible and make sense to our TAs. Make sure to clearly label each answer with the problem number you're solving so our TAs can more easily evaluate your work. As a reminder, homework is graded on effort, not correctness – please try every question!

For homework #1, you must complete the following **required** problems:

- INTR1: Haskell Setup (15 min)
- INTR2: Python Setup (15 min)
- PYTH1: Object Reference Semantics, Parameter Passing, Shallow Copying, Boxing (15 min)
- PYTH2: Duck Typing, Dunder Functions (5 min)
- PYTH3: Duck Typing, Easier To Ask For Forgiveness, Inheritance (9 min)
- PYTH4: Slicing (6 min)
- PYTH5: Inheritance, Exceptions (9 min)
- PYTH6: Interpreted vs Compiled Languages (6 min)
- PYTH7: Numpy (10 min)

- PYTH8: Class vs Instance Variables (5 min)

For homework #1, you may optionally complete the following additional problems:

- N/A

INTR1: Haskell Setup (15 min)

You can download the Haskell compiler and interpreter, GHC and GHCi, from the following link: https://www.haskell.org/ghc/download.html. On OSX and Linux, you can also use Homebrew: brew install ghc
Note in particular that the macOS distribution also requires Xcode (and its command line tools) to be installed. You can download Xcode from the App Store.

You can also find Haskell installed on the SEASNET machines if you don't want to install it on your own computer.

You have completed this task once you are able to run ghc/ghci and not get a file-not-found error.

Hello Haskell

Create a new file named *hello.hs*, and place the following expression in it:

```
greeting = "Hello, world!"
```

Now run the Haskell interpreter by typing "ghci" on your command line. You should now be

in an interactive environment known as a REPL (read-eval-print loop). Although Haskell is typically compiled, GHCi allows you to write Haskell commands and execute them as if it were an interpreted one.

You can use the `:load` command to load Haskell code in files in the working directory. Run the following command in GHCi:

```
:load hello.hs
```

Now, using the GHCi interpreter, figure out an expression that gives the length of the greeting string. You'll have completed this task once you can successfully get the length of the greeting string via typing an expression in GHCi.

INTR2: Python Setup (15 min)

Next, we'll ensure you have a working version of Python 3. Some OS distributions (Linux, macOS) already have Python 3 bundled in. To check, the following commands can be used to locate it:

macOS/Linux:

```
which python3
```

Windows:

```
where python3.exe
```

If Python is already installed, then run it to determine the version. Your version should be 3.9 or greater.

If you don't have Python installed, you can download the latest Python 3 distribution for

your OS here: https://www.python.org/downloads/.

Once you have installed Python3 v3.9 or higher and can print “Hello world!”, you will have completed this task.

PYTH1: Object Reference Semantics, Parameter Passing, Shallow Copying, Boxing (15 min)

Consider the following Python scripts:

Part A: (5 min) Here’s our first script:

```
class Box:
  def __init__(self, value):
    self.value = value

def quintuple(num):
  num *= 5

def box_quintuple(box):
  box.value *= 5

num = 3
box = Box(3)

quintuple(num)
box_quintuple(box)
print(f"{num} {box.value}")
```

Running this script yields the output:

3 15

Explain, in detail, why `box`'s `value` attribute was mutated but `num` was not. Your answer should explicitly mention Python's parameter passing semantics.

Hint: Check out the Python documentation to understand where class instances store their attributes.

Your answer:

Part B: (10 min) Here's our second script, taken from a previous CS131 midterm exam:

```
class Comedian:
  def __init__(self, joke):
    self.__joke = joke

  def change_joke(self, joke):
    self.__joke = joke

  def get_joke(self):
    return self.__joke

def process(c):
 # line A
 c[1] = Comedian("joke3")
 c.append(Comedian("joke4"))
 c = c + [Comedian("joke5")]
 c[0].change_joke("joke6")

def main():
 c1 = Comedian("joke1")
 c2 = Comedian("joke2")
```

```
com = [c1,c2]
process(com)
c1 = Comedian("joke7")
for c in com:
 print(c.get_joke())
```

Part B.i: Assuming we run the main function, what will this program print out?

Your Answer:

Part B.ii: Assuming we removed the comment on line A and replaced it with:

```
c = copy.copy(c)        # initiate a shallow copy
```

If we run the main function, what would the program print?

Your Answer:

PYTH2: Duck Typing, Dunder Functions (5 min)

(5 min.) Consider the following output from the Python 3 REPL:

```
>>> class Foo:
...   def __len__(self):
...     return 10
...
```

```
>>> len(Foo())
10
>>> len("blah")
4
>>> len(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

Explain why Python allows you to pass an object of type `Foo` or `str` to `len`, but not one of type `int`. Your answer should explicitly reference Python's typing system.

Your answer:

PYTH3: Duck Typing, Easier To Ask For Forgiveness, Inheritance (9 min)

Consider the following `Duck` class, and two functions that check whether or not an object is a `Duck`:

```
class Duck:
  def __init__(self):
    pass # Empty initializer
```

```
def is_duck_a(duck):
  try:
```

```
    duck.quack()
    return True
  except:
    return False
```

```
def is_duck_b(duck):
  return isinstance(duck, Duck)
```

Part A: (2 min.) Write a **new** class such that if we passed an instance of it to both functions, `is_duck_a` would return `True` but `is_duck_b` would return `False`.

Your answer:

Part B: (2 min.) Write a **new** class such that if we passed an instance of it to both functions, `is_duck_a` would return `False` but `is_duck_b` would return `True`.

Your answer:

Part C: (5 min.) Which function is more Pythonic: `is_duck_a` or `is_duck_b`? This reference may help provide some insight.

Your answer:

PYTH4: Slicing (6 min)

Part A: (3 min.) Consider the following function that takes in a list of integers and another integer k and returns the largest sum you can obtain by summing k consecutive elements in the list. Fill in the blanks so the function works correctly.

Example:

`largest_sum([3,5,6,2,3,4,5], 3)` should return 14.

`largest_sum([10,-8,2,6,-1,2], 4)` should return 10.

```
def largest_sum(nums, k):
  if k < 0 or k > len(nums):
    raise ValueError
  elif k == 0:
    return 0

  max_sum = None
  for i in range(len(nums)-k+1):
    sum = 0
    for num in _________________:
      sum += num
    if ____________________________________:
      max_sum = sum
  return max_sum
```

Your answer:

Part B: (3 min.) The function in part a) runs in $O(nk)$ time where n is the length of `nums`, but we can use the **sliding window technique** to improve the runtime to $O(n)$.

Imagine we know the sum of k consecutive elements starting at index `i`. We are interested in the next sum of k consecutive elements starting at index `i+1`. Notice that the only difference between these two sums is the element at index `i` (which becomes excluded) and the element at index `i+k` (which becomes included).

We can compute each next sum by subtracting the number that moved out of our sliding window and adding the one that moved in. Fill in the blanks so the function works correctly.

```
def largest_sum(nums, k):
  if k < 0 or k > len(nums):
    raise ValueError
  elif k == 0:
    return 0

  sum = 0
  for num in _________________:
    sum += num

  max_sum = sum
```

```
  for i in range(0, len(nums)-k):
    sum -= _________________
    sum += _________________
    max_sum = max(sum, max_sum)
  return max_sum
```

Your answer:

PYTH5: Inheritance, Exceptions (9 min)

We're going to design an event-scheduling tool, like a calendar. Events have a `start_time` and an `end_time`. For simplicity, we will model points in time using `int`s that represent the amount of seconds past some reference time (normally, we would use the Unix epoch, which is January 1, 1970).

Part A: (3 min.) Design a Python class named `Event` which implements the above functionality. Include an initializer that takes in a `start_time` and `end_time`. If `start_time >= end_time`, raise a `ValueError`.

The following code:

```
event = Event(10, 20)
print(f"Start: {event.start_time}, End: {event.end_time}")

try:
```

```
  invalid_event = Event(20, 10)
  print("Success")
except ValueError:
  print("Created an invalid event")
```

should print:

```
Start: 10, End: 20
Created an invalid event
```

Your answer:

Part B: (3 min.) Write a Python class named `Calendar` that maintains a private list of scheduled events named `__events`.

It should:

- Include an initializer that takes in no parameters and initializes `__events` to an empty list.
- Have a method named `get_events` that returns the `__events` list.
- Have a method named `add_event` that takes in an argument `event`:
  - If the argument is not of type `Event`, do nothing and raise a `TypeError`.
  - Otherwise, add the event to the end of the `events` list.

The following code:

```
calendar = Calendar()
print(calendar.get_events())
```

```
calendar.add_event(Event(10, 20))
print(calendar.get_events()[0].start_time)
try:
  calendar.add_event("not an event")
except TypeError:
  print("Invalid event")
```

should print:

```
[]
10
Invalid event
```

Your answer:

Part C: (3 min.) Consider the following subclass of `Calendar`:

```
class AdventCalendar(Calendar):
  def __init__(self, year):
    self.year = year

advent_calendar = AdventCalendar(2022)
print(advent_calendar.get_events())
```

Running this code as-is (assuming you have a correct `Calendar` implementation) yields the following output:

```
AttributeError: 'AdventCalendar' object has no attribute
'__events'. Did you mean: 'get_events'?
```

Explain why this happens, and list two different ways you could fix the code **within the class** so the snippet instead prints:
`[ ]`

Your answer:

PYTH6: Interpreted vs Compiled Languages (6 min)

Suppose we have 2 languages. The first is called I-Lang and is an interpreted language. The interpreter receives lines of I-Lang and efficiently executes them line by line. The second is called C-Lang and is a compiled language. Assume that it takes the same time to write an I-Lang script as a C-Lang program if both perform the same function.

Part A: (2 min.) Suppose we have an I-Lang script and a C-Lang executable that functionally perform the exact same thing. If we execute both at the same time, which do you expect to be faster and why?

Your answer:

Part B: (2 min.) Suppose Jamie and Tim are two equally competent students developing a web server that sends back a simple, plaintext HTML page. Jamie writes her server in I-Lang, whereas Tim writes his in C-Lang. Assuming that the server will be deployed locally, who will most likely have the server running first, and why?

Your answer:

Part C: (2 min.) Jamie and Tim have a socialite friend named Connie who uses a fancy new SmackBook Pro. The SmackBook Pro has a special kind of chip called the N1, which has a proprietary machine language instruction set (ISA) completely unique to anything else out there. If you are familiar with Rosetta, assume the SmackBook Pro has **no** built-in emulator/app translator. Jamie and Tim have less-fancy computers with Intel chips.

Connie has a native copy of the I-Lang interpreter on her computer. Jamie sends Connie her web server script, and Tim sends Connie his pre-compiled executable. Will Connie be able to execute Jamie's script? What about Tim's executable?

Your answer:

PYTH7: Numpy (10 min)

(10 min.) Consider the following code snippet that compares the performance of matrix multiplication using a hand-coded Python implementation versus using numpy:

```
import numpy as np
import time
from random import randint

def dot_product(a, b):
  result = 0
  for a_i, b_i in zip(a, b):
    result += a_i * b_i
  return result

def matrix_multiply(matrix, vector):
  return [dot_product(row_vector, vector) for row_vector in matrix]

# Generate a 1000 element vector, and a 1000 x 1000 element matrix
vector = [randint(0, 10) for _ in range(1000)]
matrix = [[randint(0, 10) for _ in range(1000)] for _ in range(1000)]

# Create numpy arrays
np_vector = np.array(vector)
np_matrix = np.array(matrix)
```

```
# Multiply the matrix and vector (using our hand-coded implementation)
start = time.time()
matrix_multiply(matrix, vector)
end = time.time()

# Multiply the matrix and vector (using numpy)
np_start = time.time()
np_matrix.dot(np_vector)
np_end = time.time()

print(f"Hand-coded implementation took {end - start} seconds")
print(f"numpy took {np_end - np_start} seconds")
```

If we run this code, the `numpy` operation is invariably about 100 times faster than the call to `matrix_multiply`:

```
Hand-coded implementation took 0.043227195739746094 seconds
numpy took 0.0004067420959472656 seconds
```

Assuming that the implementations of `dot_product` and `matrix_multiply` are reasonably optimized, provide an explanation for this discrepancy in performance.

Hint: Take a look at this page from the numpy docs to understand a bit more about how the package is implemented.

Your answer:

PYTH8: Class vs Instance Variables (5 min)

(5 min.) Consider the following code snippet that uses both class variables and instance variables:

```
class Joker:
 joke = "I dressed as a UDP packet at the party. Nobody got it."

 def change_joke(self):
  print(f'self.joke = {self.joke}')
  print(f'Joker.joke = {Joker.joke}')
  Joker.joke = "How does an OOP coder get wealthy? Inheritance."
  self.joke = "Why do Java coders wear glasses? They can't C#."
  print(f'self.joke = {self.joke}')
  print(f'Joker.joke = {Joker.joke}')

j = Joker()
print(f'j.joke = {j.joke}')
print(f'Joker.joke = {Joker.joke}')
j.change_joke()
print(f'j.joke = {j.joke}')
print(f'Joker.joke = {Joker.joke}')
```

Part A: (2 min) What do you expect this program to print out?

Your answer:

Part B: (3 min) What does each print statement actually print out? Why?

Your answer: