Homework 3 - Spring 2023

- Vincent Lin
- [UID REDACTED]
- Section 1D

Question 1

```
def convert_to_decimal(bits: list[int]) -> int:
    exponents = range(len(bits)-1, -1, -1)
    nums = [bit * 2**exponent for bit, exponent in zip(bits, exponents)]
    return reduce(lambda acc, num: acc + num, nums)
```

Question 2

Question 2a

```
def parse_csv(lines: list[str]) -> list[tuple[str, int]]:
    return [(string, int(num)) for string, num in [
        line.split(",") for line in lines
    ]]
```

Question 2b

```
def unique_characters(sentence: str) -> set[str]:
    return {char for char in sentence}
```

Question 2c

```
def squares_dict(lower_bound: int, upper_bound: int) -> dict[int, int]:
    return {num: num**2 for num in range(lower_bound, upper_bound+1)}
```

Question 3

```
def strip_characters(sentence: str, chars_to_remove: set[str]) -> str:
    return "".join(char for char in sentence
                   if char not in chars_to_remove)
```

Question 4

Python uses **pass by object reference** semantics for parameter passing. This means that the value passed as an argument to any function is actually the object reference to which that value is bounded, akin to passing by pointer in languages such as C++.

So considering this part of the code:

```
def quintuple(num):
    num *= 5
num = 3
quintuple(num)
```

First we pass `num` (bound to 3) to `quintuple` as an argument that's bounded to the identically named parameter, `num`. The statement `num *= 5` is actually equivalent to `num = num * 5`. Because this is actually an **assignment statement**, we're really just rebinding what the name `num` refers to in the local scope, similar to overwriting the value of a local pointer in C++. Thus, the `num` in the global scope is unaffected and stays as 3.

In the case of `box`:

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

The statement `box.value *= 5` is equivalent to `box.value = box.value * 5`. `box` references the same `Box(3)` instance in the global scope since `box` is passed by object reference. We're still rebinding what `box.value` points to due to assignment semantics, but because `value` is bound to `box` which still persists in the global scope, we can see the change even after the function executes. This is similar to overwriting the value of a pointer member variable of an object in C++.

Question 5

Python's typing system uses **duck typing**, which is the property where identical method calls will work on all data types that define such methods. If both `a` and `b` implement the `quack` method, both `a.quack()` and `b.quack()` will work.

In Python, the built-in function `len()` actually calls the `__len__` method of its argument. This means that due to duck typing, any object that implements `__len__` has a valid `len()` call. This is true of `Foo` and `str`, but not of `int`. In fact, we can verify this for ourselves in an interactive session:

```
>>> str.__len__
<slot wrapper '__len__' of 'str' objects>
>>> int.__len__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'int' has no attribute '__len__'. Did you mean:
'__le__'?
```

Question 6

Question 6a

```python
class DuckPretender:
    def quack(self) -> None:
        pass  # Doesn't need to do anything
```

Question 6b

```python
class MallardDuck(Duck):
    pass
```

Question 6c

`is_duck_a` is more Pythonic because it abides by the idiom **EAFP** ("easier to ask for forgiveness than permission"). This goes hand in hand with **duck typing** - as long as an object implements the `quack` method ("quacks like a duck"), then it should be treated like a "duck". First we attempt to call the `quack` method without checking anything else (not "asking for permission"). If it goes through, all is well. If it fails, we catch the exception and handle it gracefully ("asking for forgiveness").

This principle is contrasted by **LBYL** ("look before you leap"). Although `is_duck_b` is not technically an example of this, explicit type checking via `isinstance()` is a common pattern in Python associated with performing checks before proceeding with the operation, and this is considered not Pythonic.

Question 7

Question 7a

```python
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 (nums[j] for j in range(i,i+k)):
            sum += num
        if max_sum is None or sum > max_sum:
            max_sum = sum
    return max_sum
```

NOTE: I know the answer probably expected `nums[i:i+k]` in the first blank, but this is actually really inefficient for large `k` because list slicing returns a copy of the subsequence. The runtime complexity would still be O(nk), but using a **generator** in its place ensures that we don't use more time and memory than we need to. I verified that this implementation still passes the test cases provided. Please don't mark me wrong.

Question 7b

```python
def largest_sum(nums, k):
    if k < 0 or k > len(nums):
        raise ValueError
    elif k == 0:
        return 0
    sum = 0
    # First find the sum of the first k elements.
    for num in (nums[j] for j in range(k)):
        sum += num
    max_sum = sum
    # Now we apply the sliding window technique.
    for i in range(0, len(nums)-k-1):
        sum -= nums[i]
        sum += nums[i+k]
        max_sum = max(sum, max_sum)
    return max_sum
```

NOTE: I used a generator comprehension in the first blank as opposed to the trivial `nums[:k]` answer for the same reason as in part (a). Please don't mark me wrong.

Question 8

Question 8a

```python
class Event:
    def __init__(self, start_time: int, end_time: int) -> None:
        if start_time >= end_time:
            raise ValueError
        self.start_time = start_time
        self.end_time = end_time
```

Question 8b

```python
class Calendar:
    def __init__(self) -> None:
        self.__events: list[Event] = []
    def get_events(self) -> list[Event]:
        return self.__events
    def add_event(self, event: Any) -> None:
        if not isinstance(event, Event):
            raise TypeError
        self.__events.append(event)
```

Question 8c

This code fails because an instance variable defined with two leading underscores e.g. `__events` is actually **name-mangled** (obfuscated) into `_ClassName__events` at class definition time. This means that the `advent_calendar` does not have an `__events` instance variable but rather a `_Calendar__events` instance variable. Note that it is named with `Calendar` because this variable is assigned within the `Calendar` class even though `advent_calendar` is an instance of `AdventCalendar`. Because `get_events()` is coded to return `self.__events`, it fails and raises an `AttributeError` citing the failed lookup of the `__events` name.

SOLUTION 1: A NAIVE fix would be to give `AdventCalendar` its own `__events` at initialization and override the getter:

```python
class AdventCalendar(Calendar):
    def __init__(self, year):
        self.year = year
        self.__events: list[Event] = []  # Added this line.
    # Added this method.
    def get_events(self) -> list[Event]:
        return self.__events
```

Obviously, this approach defeats the purpose of using inheritance in the first place and introduces error-prone code repetition.

SOLUTION 2: The BETTER fix is to call the superclass' constructor when initializing:

```python
class AdventCalendar(Calendar):
    def __init__(self, year: int) -> None:
        super().__init__()  # Added this line.
        self.year = year
```

This fixes the problem because `super().__init__()` calls `Calendar.__init__(advent_calendar)`, meaning the assignment statement `self.__events = []` within the constructor code will run and give `advent_calendar` its own `__events` attribute (which of course is then name-mangled into `_Calendar__events`).

Question 9

Yes, Python supports closures.

```python
def outer_function(x: int) -> Callable[[int], int]:
    def inner_function(y: int) -> int:
        return x + y
    return inner_function
add_6 = outer_function(6)
print(add_6(15))  # 21
```

Above is an example of a function returning another function that adds its argument to the argument of the outer function. This can only be possible if Python supports closures because `inner_function` must capture the free variable `x` from `outer_function`'s scope to be able to "remember" its value when being called from any other scope.

Question 10

Question 10a

Assuming C-Lang is compiled down to machine code, we would expect the C-Lang executable to execute faster than the I-Lang script.

Compiled programs tend to be faster than their interpreted equivalents because the former is executed as machine instructions directly on the host machine's CPU. Furthermore, if the compiler is an optimizing compiler, the programmer could choose to sacrifice some time at compile time to produce a more optimized version of the executable - this is analogous to giving a competitor "more time to prepare" such that at runtime (which we're actually measuring) it can perform much better.

On the other hand, interpreters must tokenize, parse, and execute their programs all at runtime. Furthermore, all the data structures and runtime concepts like iteration, function calls, etc. in the interpreted language would be implemented as some equivalent in the host language, and this added layer of abstraction introduces inevitable overhead.

Question 10b

Jamie will likely have the server running first. Interpreted languages like I-Lang tend to be easier to develop with because:

- A program can be up and running (executed) directly by the interpreter at any time while compiled languages must go through a compilation step before executing every time the source code changes.
- Although dependent on the language and tools available, interpreted languages tend to be easier to debug because the runtime errors would be caught by the interpreter, which could then take some helpful form of action like providing a traceback. Compiled programs on the other hand are completely detached from the compiler after being compiled. Runtime errors would be caught by the OS, which obviously has no knowledge of the source code that produced the offending executable - it only knows to do something like report a segmentation fault.

Question 10c

Connie will be able to execute Jamie's script, assuming Jamie did not write any hardware-dependent code in the script itself.

Interpreted languages have the upside that as long the interpreter is available for an OS/ISA, it will be able to run scripts unchanged. This is because the interpreter is responsible for *creating* the runtime environment for the program that is guaranteed to comply with the language specification; how it implements it i.e. maps language features to the specific hardware architecture is a matter for the writers of the interpreter, not the developers using the language.

Compiled languages on the other hand have the downside that compiled executables are only guaranteed to work on specific ISAs. This is because executables are composed of machine code, which is only compatible on architectures that interpret that binary sequence in the way the compiler intended it to. Tim or Connie would have to look for or write a compiler for C-Lang compatible with SmackBook Pro, but because the N1 chip ISA is proprietary, this would be difficult. Even then, Tim would also have to provide his source code and build instructions such that Connie can compile an executable that works on her machine, which is a major hassle compared to Jamie's script.

Question 11

The NumPy library implements much of its core functionality in C. From the attached reading:

> NumPy gives us the best of both worlds: element-by-element operations are the "default mode" when an *ndarray* is involved, but the element-by-element operation is speedily executed by pre-compiled C code.

This means that when users use a NumPy operation like `c = a * b` where `a` and `b` are `ndarray`s, the behind-the-scenes computations are being done by pre-compiled C executables, which run directly on the host machine's hardware instead of within Python's runtime. The difference in performance is much more apparent with large quantities of data like a 1000x1000 matrix in the example. Looping through such structures in Python is slow because every iteration is interpreted, so there's the natural overhead of the interpreter conducting a bunch of checks, memory management, etc. behind the scenes that slow down our actual work. This is why NumPy code is *much* faster compared to the hand-coded implementation that's written in pure Python.

Question 12

Question 12a

This program will print out:

```
j.joke = I dressed as a UDP packet at the party. Nobody got it.
Joker.joke = I dressed as a UDP packet at the party. Nobody got it.
self.joke = I dressed as a UDP packet at the party. Nobody got it.
Joker.joke = I dressed as a UDP packet at the party. Nobody got it.
self.joke = Why do Java coders wear glasses? They can't C#.
Joker.joke = How does an OOP coder get wealthy? Inheritance.
j.joke = Why do Java coders wear glasses? They can't C#.
Joker.joke = How does an OOP coder get wealthy? Inheritance.
```

Question 12b

The print statements print out the output in part (a).

The first two lines are what's printed right after `j = Joker()`. Both are the UDP joke because both `j.joke` and `Joker.joke` resolve to the class variable `joke` of `Joker`. The former might be unexpected, but it still works because attribute lookup first searches for an instance variable named `joke`, and if it's not found, it searches for a class variable named `joke`.

The next two lines are what's printed at the start of `change_joke`. These have the same output as the first two lines because `self` and `Joker` are the same object references as `j` and `Joker` from the outer scope respectively.

The next two lines are what's printed at the end of `change_joke`. `self.joke` is now the Java joke because of the prior assignment statement to `self.joke`. Due to the attribute lookup process described earlier, `joke` is found as an instance variable of `self`, so it's returned and no further lookup is performed on its class. `Joker.joke` is expectedly the inheritance joke because we explicitly set the class variable to that in the prior `Joker.joke` assignment statement.

The last two lines are what's printed after `change_joke` returns. These are the same as the previous two lines because `j` and `Joker` are the same object references as `self` and `Joker` from within `change_joke`'s scope respectively.