Homework 1 Solutions

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

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:

```python
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.

Solution:

This answer is as instructive as we can make it - this level of detail will not be required on the exam.

In Python, [everything is an object](); every variable is essentially a pointer to an object. This is true for primitive data types too (including `int`, `bool` and `str`). Furthermore, Python uses pass by object reference for function parameters: if we have a function `f` with parameter `x`, and we invoke `f` by passing some variable `a`, then `x` merely points to whatever object that `a` points to. So, for example, when we refer to `num` *inside* the `quintuple` function, it is a separate pointer to the original object that the `num` *outside* the function points to. In other words, we have two different pointers that both point to the object 3.

Next, it is important to note that when you reassign a variable to a new value, you aren't changing the value of the old object pointed to by that variable - you make the variable point to the new object you're assigning it. In the case of the expression `num *= 5` (which is [syntactic sugar]() for `num = num * 5`), we change the `num` variable *inside* of the `quintuple` function such that it now points at the object 15. We do not modify the object that it previously pointed to (the `int` 3), nor do we change what the `num` variable *outside* of the function points to (which is why 3 is still printed).

However, custom classes in Python (like the `Box` class) *do* store their attributes separately (as alluded to in the hint, they're stored in a dictionary object named `__dict__`). So the expression `box.value *= 5` modifies the `value` attribute on the same object pointed at by both `box` variables (i.e. the one inside the function *and* outside. Both variables point to the same object). This is why 15 is printed.

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?

`Answer for part i is below.`

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?

`Answer for part ii is below.`

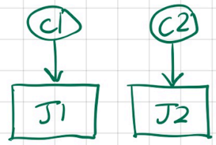
`Answers for part i and ii, as written by our former student Vincent Lin:`

- Green describes a new allocation, extension, etc. as a result of the current line.
- Red describes something that has been removed as a result of the current line.
- Blue describes what's being referenced in a method call on the current line.

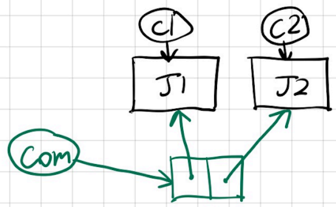
Also included are some notes in gray describing non-obvious behavior that Python performs behind the scenes.

Part (i) is below:

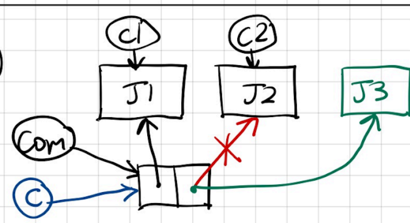
```
main():
    c1 = Comedian("joke1")
    c2 = Comedian("joke2")
```



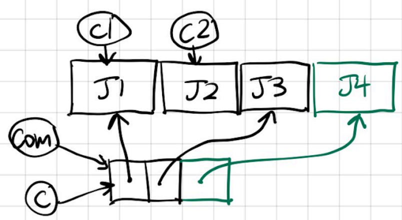

```
com = [c1, c2]
```




```
process(c)
  c[1] = Comedian("joke3")
```

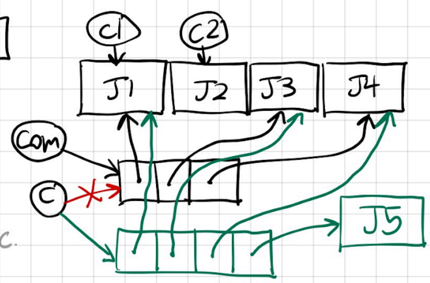



```
c.append(Comedian("joke4"))
```



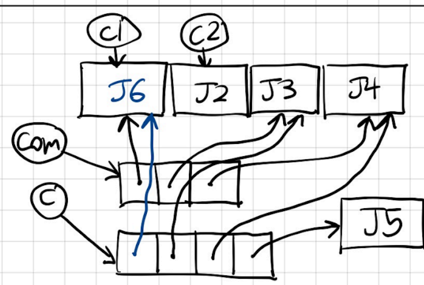

```
c = c + [Comedian("Joke5")]
```

A *new* Python list is constructed, the concatenation of the object references in c and the object reference to J5. *Then*, it is bound to the name c.



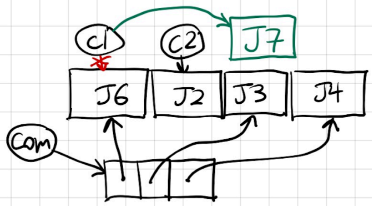

```
c[0].change_joke("joke6")
```

Remember, methods act on the *object reference*.

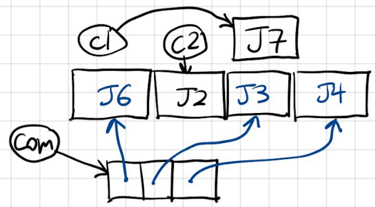


```
main():
    c1 = Comedian("joke7")
```

The local variable c goes away and J5 is GC'ed.



```
for c in com:
    print(c.get_joke())
```

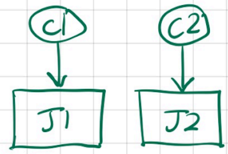


FINAL OUTPUT

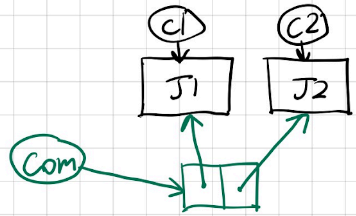
```
joke6
joke3
joke4
```

Part (ii)

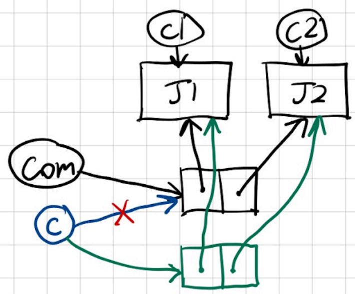
```
main():
    c1 = Comedian("joke1")
    c2 = Comedian("joke2")
```



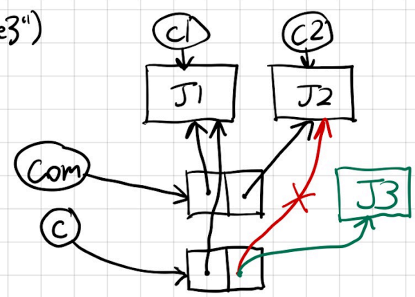
```
com = [c1, c2]
```



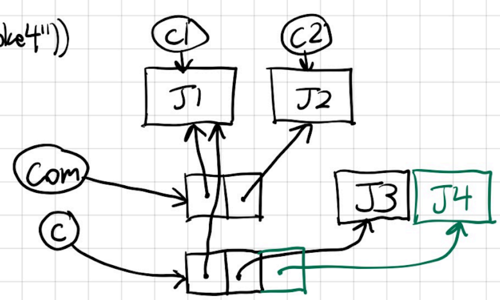
```
process(c):
    c = copy.copy(c)
```



```
c[1] = Comedian("joke3")
```

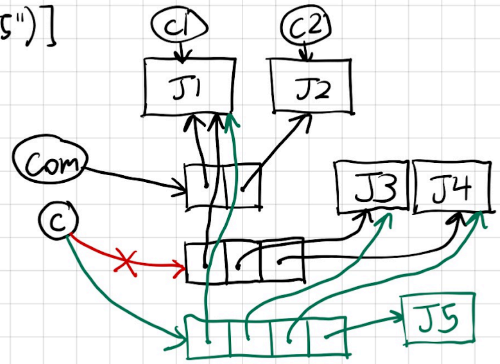


```
c.append(Comedian("joke4"))
```



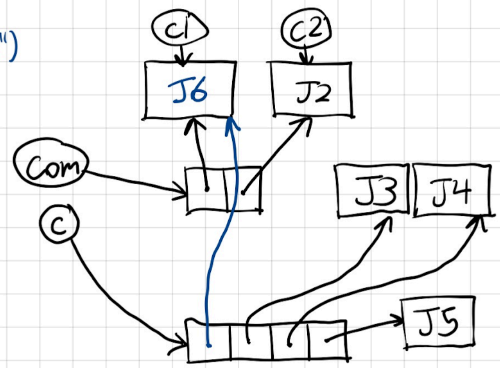
c = c + [Comedian("joke5")]

The list originally pointed by c goes away!



---

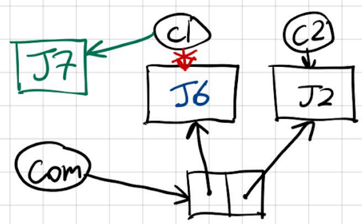
c[0].change_joke("joke6")



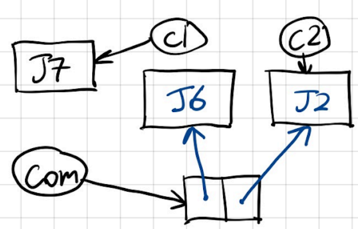
---

main():
    c1 = Comedian("joke7")

The local variable c goes away and J3, J4, J5 are GC'ed



```
for c in com:
    print(c.get_joke())
```



FINAL OUTPUT

```
joke6
joke2
```

In General, for problems like these, there's a very simple trick: always, always draw pictures. You are NOT the interpreter. You are a human. Always draw pictures. Feel free to use these conventions:

- Boxes represent values, actual objects on the heap.
- Ellipses represent variables, names on the stack that are *bound* to the objects.

Drawing pictures effectively reduces the problem to following a bunch of arrows. This way, you'll *never* get lost.

The key part of this question is that it's testing object reference semantics. Remember that while Python made the syntax prettier, everything is really a pointer under the hood, *including* lists - a list is itself a pointer to an array of more pointers, which *then* point to the objects "stored" in the list.

Whenever you call a method like `c.append`, `some_comedian.change_joke`, etc. follow the arrow coming out of the ellipse with the variable name and perform the operation on the reference object(s). This works for more complex accesses like `c[0].change_joke`:

1. First follow the arrow out of `c`.
2. Choose the arrow out of array position `0`.
3. Follow that arrow and call `change_joke` on that object.

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.

Solution:

Python is duck-typed, which means that the type of an object is less important than the attributes and methods it defines. For example, functions in Python do not discriminate by requiring you to pass objects of a certain type - you can pass in anything you'd like. However, if you try to invoke a method or reference an attribute that does not exist on an object, Python will raise an error.

Built-in functions work the same way. In the example, you can see that for the `len` function to return without throwing an exception, the caller merely needs to pass in an object that has the `__len__` function defined. In other words, the implementation of `len("blah")` at some point calls `"blah".__len__()`. `int` objects do not have a `__len__()` function, so the built-in `len` raises a `TypeError`.

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`.

Solution:

```
class RubberDuck:
  def quack(self):
    print("Squeak")
```

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`.

Solution:

```python
class Mallard(Duck):
  pass # empty class, no new added methods/fields
```

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

Solution:

`is_duck_a` is more Pythonic. In fact, both approaches have canonical names: "Easier to ask for forgiveness than permission" (EAFP) and "Look before you leap" (LBYL) respectively. Essentially, in Python it's considered more robust to make assumptions (e.g. that a Duck has a `quack()` method) and handle errors when those assumptions do not hold than to assert conditions ahead of time (am I dealing with a Duck object?) and carry on assuming they will always be true. For a concrete example of why, check out this old email from the Python mailing list.

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`.

```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 _______________:
      sum += num
    if _________________________________:
      max_sum = sum
  return max_sum
```

Solution:

```
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[i:i+k]:
      sum += num
    if max_sum is None or sum > max_sum:
      max_sum = sum
  return max_sum
```

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
```

Solution:

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

```
sum = 0
for num in nums[0:k]:
  sum += num

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

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 `ints` 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
```

Solution:

```python
class Event:
  def __init__(self, start_time, end_time):
    if start_time >= end_time:
      raise ValueError
    self.start_time = start_time
    self.end_time = end_time
```

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:

```python
calendar = Calendar()
print(calendar.get_events())
calendar.add_event(Event(10, 20))
print(calendar.get_events()[0].start_time)
```

```python
try:
  calendar.add_event("not an event")
except TypeError:
  print("Invalid event")
```

should print:

```
[]
10
Invalid event
```

Solution:

```python
class Calendar:
  def __init__(self):
    self._events = []

  def get_events(self):
    return self._events

  def add_event(self, event):
    if not isinstance(event, Event):
      raise TypeError
    self._events.append(event)
```

Notice that we used `isinstance` even though, in the solution to question 6, we argued that it is more Pythonic to ask for forgiveness than to look before you leap. This would be the only way to check if an object is truly of a particular type. An alternative approach we could have taken would be to require that any object passed in implements the same set of methods as the Event class, rather than require only Event (or subclasses of Event) objects be passed in. This would enable duck typing to be used.

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:

```
[]
```

Solution:

While the `AdventCalendar` class does inherit from the `Calendar` class, it does not explicitly invoke its base class' initializer via `super().__init__`. Because of this, `Calendar`'s initializer (which adds an empty `events` list to the instance) is never run. Notice, however, that we do still inherit the methods defined on `Calendar` (namely `get_events`).

To fix this, inside `AdventCalendar`'s `__init__` function we can either add an explicit call to the super initializer:

```
super().__init__()
```

or we can just add an `events` attribute ourselves. Of course, calling the superclass initializer would be better practice (since it leads to less repetitive code, especially in cases where the base class initializer has additional logic).

```
self._events = []
```

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?

Solution:

We would expect the C-Lang executable to be faster. The C-Lang program is compiled directly to machine code, whereas the interpreted I-Lang code is translated into lower-level representations at run-time. This extra step creates additional overhead which will inevitably cause it to be slower than the pre-compiled code.

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?

Solution:

We would expect Jamie to get the web server running first. From start to finish, the only difference between both development processes is that Jamie needs to execute her code using the interpreter, whereas Tim needs to compile his into an executable. In general, the compilation process takes a nontrivial amount of time, whereas the I-Lang interpreter can begin executing lines of code right away.

It could also be correct to argue that Tim would be able to get the server running first. If the I-Lang interpreter was extremely slow, or if the C-Lang compiler was abnormally fast, then as in part a), the additional overhead associated with interpreting each line of I-Lang may be enough to make Tim the winner. For most modern languages, however, compilation (which involves preprocessing, compiling, assembling, and linking) is a process that takes much longer than simply interpreting lines of code.

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?

Solution:

Connie will be able to execute Jamie's script because she already has a working version of the I-Lang interpreter on her computer. However, she will not be able to run Tim's executable, which has already been compiled into machine code that is tailored for Intel's ISA - it isn't portable.

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)
```

```python
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.

Solution:

The key to answering this question is to recognize that under the hood (as referenced by the hint), numpy's implementation (which includes functions like `dot`) heavily relies on pre-compiled, optimized C code. There are therefore two main reasons (only the first of which we would expect you to reasonably elaborate on on an exam) that numpy is significantly faster.

Firstly, as discussed in previous questions, compiled code is much more performant than interpreted code, because the latter involves the overhead of translation to

lower-level representations at runtime. This is especially pertinent for our code, which will call the `dot_product` function at least 1000 times. For each invocation, the Python interpreter will convert the bytecode corresponding to `dot_product` into machine-executable instructions. This process has already been done ahead of time for the C code.

Secondly, the underlying memory representation for numpy arrays is much more efficient than that of Python lists. Recall that in Python, everything is an object, which means that the matrix variable actually contains one million pointers. Not only is this less space efficient, there is the chance of poor spatial locality: the objects pointed to in the matrix might be scattered throughout RAM. However, numpy provides support for contiguous arrays where each element is actually next to each other in memory (as in C). This enables more efficient caching techniques at the CPU level and chip-specific optimizations such as SIMD.

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?

Solution:

#note: newlines added for clarity

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.

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

Solution:

The correct answer is above, so this part will just be an explanation of the above.

Note that modifying a class variable on the class namespace, modifies all the instances of the class.

Also, for the field `self.joke`, if there is no instance variable `joke` in the `self` instance, then `self.joke` will reference the class variable `joke` on the class that `self` is an instance of (in this case, Joker). However, if an instance variable `self.joke` exists, it will shadow the class variable of the same name, overriding and hiding it.

The line:

```
self.joke = "Why do Java coders wear glasses? They can't C#."
```

creates a new `joke` variable that gets added to the instance of the Joker class.

So, the first 4 lines are the same, since `self.joke` and `Joker.joke` refer to the same class variable.

Then, line 5 refers to the (newly created!) instance variable, and line 6 refers to the changed class variable.

Finally, line 7 refers to the instance variable and line 8 refers to the class variable, so the same situation as lines 5 and 6.