Practice with While Loops

Today starts as a Paper + Pencil or Tablet + Pencil day... please keep laptops stowed away!

COMP110 - CL10

2024/02/27

Announcements

- EX03 - Wordle
  - Uses concepts through LS07
  - Implement the popular online game Wordle!
  - Due 3/1 at 11:59pm

- Quiz 1
  - Median 88%

- Quiz 2 - Thursday 3/7

Warm-up

Trace a Memory Diagram

```python
def swap(t: tuple[int, ...], i: int, h: int) -> None:
    """Swap item at index i with item at index h"""
    assert i < len(t) and h < len(t)
    temp: int = t[i]
    t[i] = t[h]
    t[h] = temp


values: tuple[int, ...] = (10, 20, 30)
swap(values, 0, 2)
print(values)
```

Tuples are Immutable!

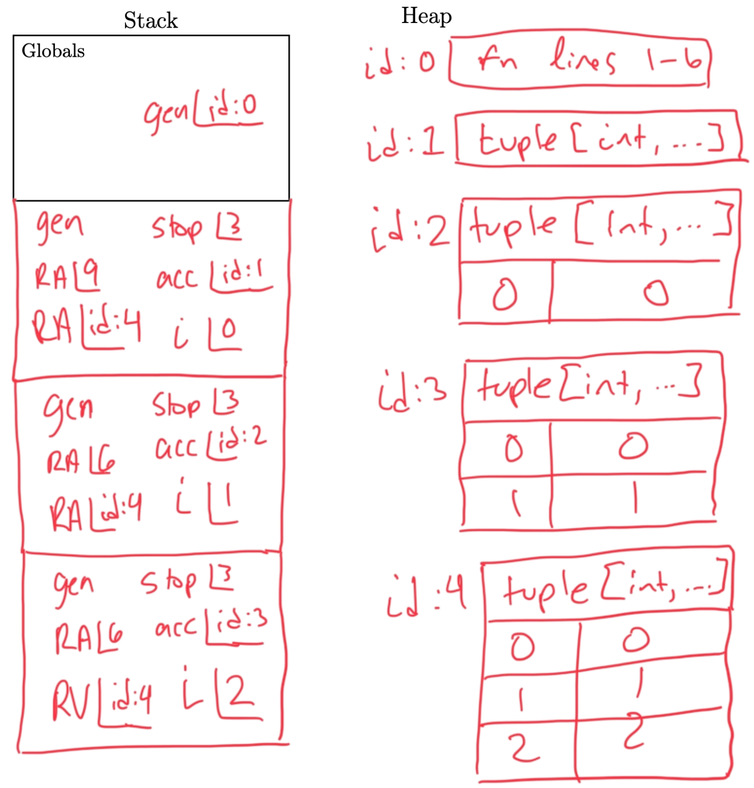
Recall: Practice Quiz 1 - Question 3

Question 3: Memory Diagram Trace a memory diagram of the following code listing and then answer the sub-questions. You do not need to diagram the sub-questions.

```
1  def gen(stop: int, acc: tuple[int, ...] = (), i: int = 0) -> tuple[int, ...]:
2    """Generate a tuple from i to stop."""
3    if i >= stop - 1:
4      return acc + (i,)
5    else:
6      return gen(stop, acc + (i,), i + 1)
7
8
9  print(gen(3))
```

* Each tuple concatenation produces a new tuple object on the heap

- Once a Tuple object is constructed, its items cannot be changed or removed and new items cannot be added
- Immutability makes Tuples simpler to reason about!
- However, as the gen function demonstrates, they are memory/space inefficient for algorithms that produce large tuples or need to update/remove items with any frequency.



Lists are Mutable Sequences in Python

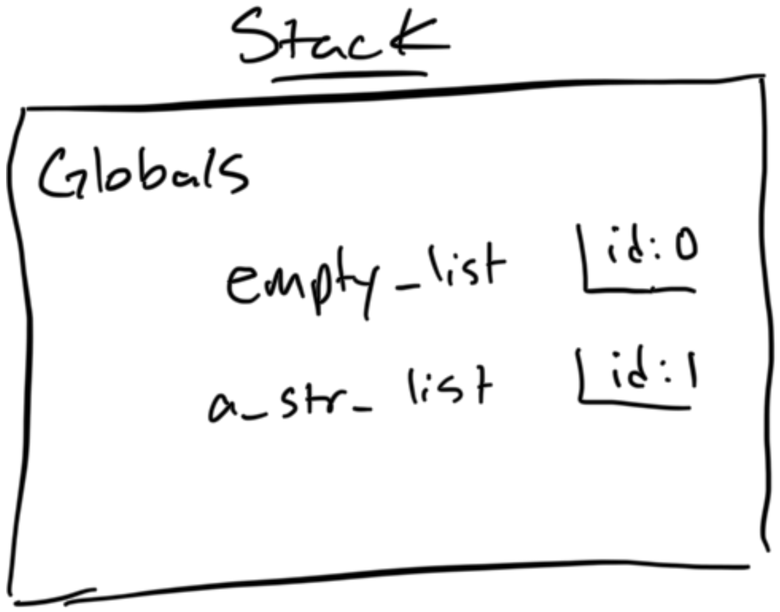
Sequences are ordered, 0-indexed collections of values

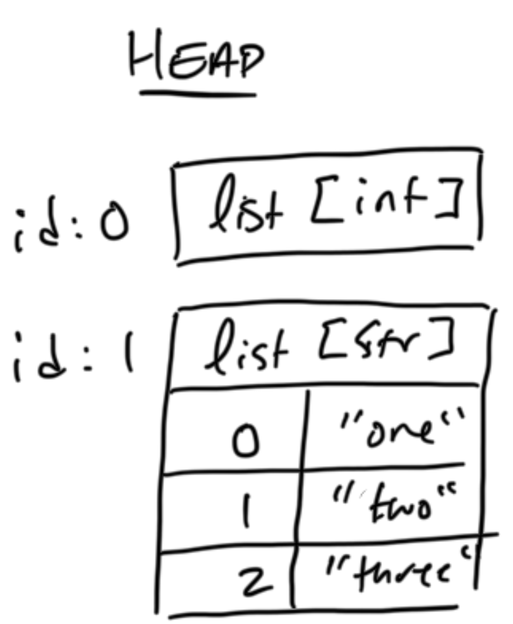
Feature	Syntax	Purpose
Type Declaration		
Constructor (function)		
List Literal		
Access Value		
Assign Item		
Length of List		

List are Heap Objects in Memory Diagrams

```python
"""Examples of list representation on Heap"""

empty_list: list[int] = []
a_str_list: list[str] = ["one", "two", "three"]
```





Trace a Memory Diagram

```
1   def swap(t: list[int], i: int, h: int) -> None:
2       """Swap item at index i with item at index h"""
3       assert i < len(t) and h < len(t)
4       temp: int = t[i]
5       t[i] = t[h]
6       t[h] = temp
7
8
9   values: list[int] = [10, 20, 30]
10  swap(values, 0, 2)
11  print(values)
```

Warning: Mutate Reference Arguments with Caution

Mutating arguments can be an unwelcome surprise to the function caller!

- The previous example demonstrated a surprising outcome:
  - The function returned None, and yet...
  - The *global* object named *values* was mutated by a function call!
- References (id:X values) to mutable heap objects (e.g. lists) make this possible
  - With great power, comes great responsibility! Always document parameters that will be mutated in the docstring!
- Wherever possible, avoid mutation... we'll see a strategy for how to do so shortly!

```
1   def swap(t: list[int], i: int, h: int) -> None:
2       """Swap item at index i with item at index h"""
3       assert i < len(t) and h < len(t)
4       temp: int = t[i]
5       t[i] = t[h]
6       t[h] = temp
7
8
9   values: list[int] = [10, 20, 30]
10  swap(values, 0, 2)
11  print(values)
```

Methods Calls

Methods are special functions shared by objects of the same class.

- In LS03, you read about Expressions and encountered a Method Call Expression:
  - `"hello, world".upper()` is a method call that evaluates to `"HELLO, WORLD"`
  - If variable `a_str` is bound to `"hi"`, then `a_str.upper()` evaluates to `"HI"`
- Method call syntax is `object_expr.method_name(argument_0, ...)`
  - Example: `a_list.append("hello")`
- A method call is like a function call *directed at and carried out upon a particular object*
  - We will learn how to define classes and methods in the next unit!

Common, Useful Methods of the List Class

Feature	Method Name	Example
Append item to a List	append	
Remove item from a List	pop	
Copy a List	copy	
Count Items in a List	count	
Reverse a List	reverse	
Find Index of a Value	index	
Clear Items of a List	clear	

Trace a Memory Diagram

```
1   def gen(stop: int) -> list[int]:
2       """Generate a list from 0 to stop, not inclusive."""
3       i: int = 0
4       acc: list[int] = []
5       while i < stop:
6           acc.append(i)
7           i = i + 1
8       return acc
9
10
11  print(gen(3))
```

Compare

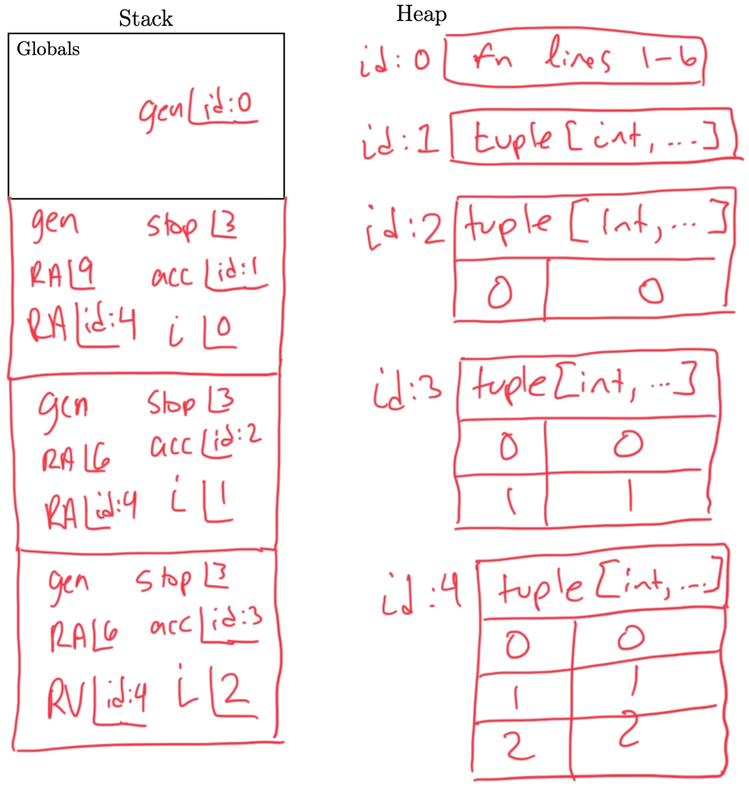
Recall: Practice Quiz 1 - Question 3

Question 3: Memory Diagram Trace a memory diagram of the following code listing and then answer the sub-questions. You do not need to diagram the sub-questions.

```
1  def gen(stop: int, acc: tuple[int, ...] = (), i: int = 0) -> tuple[int, ...]:
2    """Generate a tuple from i to stop."""
3    if i >= stop - 1:
4      return acc + (i,)
5    else:
6      return gen(stop, acc + (i,), i + 1)
7
8
9  print(gen(3))
```

* Each tuple concatenation produces a new tuple object on the heap

- Compare the diagram we just made with the equivalent diagram using immutable tuples and recursion.



Why have both Immutable *and* Mutable objects?

- A big, deep question in learning computer science fundamentals!

- Solution design trade-offs:
  - Immutable objects are easier to reason about and less error-prone to work with
  - Mutable objects tend to be more space (memory) *and* time (cost of operations) efficient for many problems

- Middle-ground:
  - Implement **pure** functions that treat mutable objects as immutable *until your problem/algorithm demands maximal efficiency*

Trace a Memory Diagram

```
1   def swap(t: list[int], i: int, h: int) -> list[int]:
2       """Returns a copy of t with indices i and h swapped"""
3       assert i < len(t) and h < len(t)
4       copy: list[int] = t.copy()
5       temp: int = copy[i]
6       copy[i] = copy[h]
7       copy[h] = temp
8
9
10  values: list[int] = [10, 20, 30]
11  result: list[int] = swap(values, 0, 2)
12
13  print(values)
14  print(result)
```