CS 134 Lecture 12: Mutability
Announcements & Logistics

- **HW 5** due Mon March 4 at 10 pm on GLOW
- **Lab 4** Part 1 autograded feedback and Lab 3 feedback will be released today
- Reminder that Midterm is **March 14**
  - Two exam slots: 6-7.30 pm, 8-9.30 pm
  - Room: Bronfman auditorium
- Midterm review Monday March 11 evening 7-9 pm in Bronfman
- How to study: review lectures
  - Practice past HW and labs on pencil and paper
  - Additional POGIL worksheets posted on course website (resources)

Do You Have Any Questions?
Last Time

- New iteration statement: the **while** loop
  - "Conditional" looping statement
  - Useful when we don't know a sequence or stopping condition ahead of time
Today's Plan

• Mutability and its consequences: aliasing
Mutability
Lists are a **mutable** data type in Python:

- After a list is created, we can **change** its value
- There are **many ways** to mutate a list, we will only discuss two of these
  - Direct assignment (e.g., `lst[index] = item`)
  - Appending to list using `.append(item)` notation
Direct Assignment

• An assignment operation on an **existing index** of a list changes the value stored at that index

Syntax:  `my_list[index] = item`

```python
>>> my_list = ['cat', 'dog']
```

```python
>>> my_list[1] = 'fish'
```

```python
>>> my_list
['cat', 'fish']
```

```python
>>> my_list[7] = 'oops'
IndexError: list assignment index out of range
```

Can only assign to **existing** indices
Appending Items to List

- We can **mutate a list** by appending an item to it.
  - Places the new item *after* the current last item, increasing length by 1.

**Syntax:** `my_list.append(item)`

```
>>> my_list = ['cat', 'dog']
>>> my_list.append('fish')
>>> my_list
['cat', 'dog', 'fish']
```

**Important:** No `[]` around item!
Sneaky Appending

- We've often updated "accumulator lists" by "appending" items in loops
- So far we have been using `+= (concatenation)`
  - `var += val` normally is a shorthand for `var = var + val`
  - But when `var` is a list, Python *secretly* calls `var.append(val)`

```python
>>> my_list = ['cat', 'dog']
>>> my_list += ['fish']
>>> my_list
['cat', 'dog', 'fish']
```

Python actually replaces `+=` with `append` without telling us!
Explicit Appending

• If we instead explicitly use the `.append(item)` syntax, then the code we execute is the code that we actually wrote.

• This also avoids one of the recurring errors that we've been running into in our labs! (Type mismatches with `+=`)

```python
>>> my_list = ['cat', 'dog']
>>> my_list += ['fish']
>>> my_list
['cat', 'dog', 'fish']
```

Brackets are **needed** here because we are adding (`+`) a list (`my_list`) to another list (`['fish']`)

```python
>>> my_list = ['cat', 'dog']
>>> my_list.append('fish')
>>> my_list
['cat', 'dog', 'fish']
```

**NO** brackets needed here because we are passing the item we want to append (`'fish'`) as an argument to the append method (special type of function)
Appending to Accumulate in a List

- We need to be careful about the type of item we provide to append.

Syntax: `my_list.append(item)`

If item is a list, then the entire list is appended.

```python
>>> my_list = ['apple', 'orange', 'banana']

>>> my_list.append(['peach'])

>>> my_list
['apple', 'orange', 'banana', ['peach']]
```

You may use `.append()` instead of `+=` in Lab 4 because they are equivalent in Python, but no other list/string "dot methods".
We have discussed the following types in class:

- `int`, `float`, `Boolean`, `string`, `list`, `range()`

Python is an object-oriented language

- Everything in Python is an **object** and has a **type**

Each type has **methods** you can call on objects of that type, e.g.,

- string objects have `.find()`, `.format()`, `.split()`, etc
- list objects have `.append()`, `.extend()`, etc

We have intentionally not discussed these in class so far (will do so later)

For lists, we are introducing `.append()` method as this is already being used "behind the scenes" with `+=`
Other data types we have seen are **immutable**

- Strings, ints, floats, range() are immutable data types
- Once created, we **cannot** change the value of an immutable data type

```python
>>> my_string = 'cat'
>>> my_string[0] = 'b'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[25], line 2
  1 my_string = 'cat'
----> 2 my_string[0] = 'b'
TypeError: 'str' object does not support item assignment
```

Will this let us change `my_string` to 'bat'? **Cannot change a string!**
Mutability has Consequences!

• Mutability of data types can have **unintended consequences**
• Consider the Python code on the left (involving **strings** which are **immutable**) vs right (involving **lists** which are **mutable**)

```python
>>> word = "hello"
>>> copy = word
>>> word = word + "world"

Changing **word** does not change **copy**
```
```
>>> word_list = ["hello"]
>>> copy = word_list
>>> word_list.append("world")

Changing **word_list** also changes **copy**
```
```
```
>>> word = word + "world"
```
```
```
```
```
```
>>> copy
['hello', 'world']
```
```
```
```
Aliasing:
Side-effect of Mutability
What is the difference between a clone and an alias?

- Clones appear the same but are actually different objects.
- Alias is another name for the same object.

To define whether something is a clone or alias in Python, we need to revisit variables and how their values are stored "under the hood".
Consider an assignment operation such as `num = 5`.

The variable **name** `num` is a way to refer to a unique address in memory where the **value** 5 is stored.

- This address is called the **identity** of this object.

```python
>>> num = 5
```

**Identity of num**: memory address where 5 is stored (e.g., 0x4486937008)

**Value of num**: 5
Value vs Identity

- An **object’s identity** never changes once it has been created.
- On the other hand, an **object’s value** may be changeable.
  - Objects whose values can change are called **mutable**.
  - Objects whose values cannot change are called **immutable**.

```python
>>> num = 5
```

Variable names like `num` point to memory addresses of stored value.

Memory address: `0x4486937008`
Clone and Alias in Python

- A **clone** of an object has the *same value* but *different identities*
  
  - Mutating a clone does not change the original object

- An **alias** of an object has the *same value* and the *same identity*
  
  - Mutating an alias also mutates the original object

Different identities (locations in memory)  
Same identity (same location in memory)
Clones and Aliases in Python

- Giving a new name to an existing immutable object creates a clone.
- Giving a new name to an existing mutable object creates an alias.

```python
>>> word = "hello"
>>> copy = word
>>> word = word + "world"
>>> copy
"hello"
```

```python
>>> word_list = ["hello"]
>>> copy = word_list
>>> word_list.append("world")
>>> copy
["hello", "world"]
```

**copy** is a **clone** of **word**, changing word does not change **copy**

**copy** is an **alias** of **word_list**, changing word changes **copy**
Strings are Immutable

```python
>>> word = "hello"
>>> copy = word
```

`copy` is a clone of `word`
Strings are Immutable

```python
>>> word = "hello"
>>> copy = word
>>> word = word + "world"
>>> copy
"hello"
```

```
word

'hello'

'thewllo world'

Instead of mutating `word`, create a new object with a different identity and value

changing `word` does not change `copy`
```

Attempts to change an immutable object create a new object
Attempts to change an immutable object create a clone

>>> num = 5
>>> num = num + 1

Trying to change the value of `num` creates a **new object** with a different identity
List Aliasing

- Any assignment or operation that creates a new name for an existing mutable object implicitly creates an alias

```python
>>> word_list = ['hello']
>>> copy = word_list
```

Since a list is mutable, we are not creating a clone, but rather an alias
List Aliasing

• Any assignment or operation that creates a new name for an existing **mutable object** implicitly creates an **alias**

```python
>>> word_list = ['hello']
>>> copy = word_list
>>> word_list.append('world')
>>> copy
['hello', 'world']
```

Changing `word_list` changes `copy`
Summary: Mutability in Python

**Strings, Ints, Floats are Immutable**
- Once you create them, their value cannot be changed
- Referring to these objects by a new variable name creates a clone
- All expressions that manipulate these objects yield a new object. They do not modify the original object

**Lists are Mutable**
- List values can be changed
  - Can mutate a list (using direct assignment or `.append()`) 
- Attempts to refer to a list by a new variable name creates an alias
How to Avoid Aliasing Side-effects
Using Immutable Types

• Aliases are never created for immutable data types

• We can safely make clones and not worry about accidentally modifying the original

• Thus any operation on strings, ints, or floats is safe from aliasing
  • Sequence operations such as slicing ([start:end]) and concatenation (+) always create new strings as it is impossible to mutate strings

• We will see an immutable alternative to lists next week
  • tuple (an immutable sequence)
Avoiding Aliasing with Lists

- When using lists, we can avoid aliasing by being careful
- An assignment of a literal value (i.e., an expression with no variables) to a variable creates a new object
- An assignment of a new list (i.e., an expression enclosed with [])) to a variable creates a new object
- var = [item] always creates a new list

```python
>>> list1 = [1, 2, 3]
>>> list2 = list1
>>> my_lst = [1, 2, 3]
```
Sequence Operations on Lists

• We can force Python to create a clone of a list instead of an alias by using sequence operations.
• Sequence operations such as slicing `[:]` and concatenation `(+)` on lists create **new lists**.
  • They do not create an alias or mutate the original list.

```python
>>> nums = [42, 11]
```
• We can force Python to create a clone of a list instead of an alias by using sequence operations
• Sequence operations such as slicing [:] and concatenation (+) on lists create **new lists**
  • They do not create an alias or mutate the original list

```python
>>> nums = [42, 11]
>>> nums = nums + [3]
```
Sequence Operations on Lists

• We can force Python to create a clone of a list instead of an alias by using sequence operations.

• Sequence operations such as slicing [:] and concatenation (+) on lists create **new lists**.
  • They do not create an alias or mutate the original list.

```python
>>> nums = [42, 11]
```
Sequence Operations on Lists

• We can force Python to create a clone of a list instead of an alias by using sequence operations

• Sequence operations such as slicing `[: ]` and concatenation `( + )` on lists create **new lists**

  • They do not create an alias or mutate the original list

```python
>>> nums = [42, 11]
>>> new = nums[:]

new is a clone but not an alias!
```
Sequence Operations on Lists

- We can force Python to create a clone of a list instead of an alias by using sequence operations.
- Sequence operations such as slicing `[:]` and concatenation `+` on lists create **new lists**.
  - They do not create an alias or mutate the original list.

```python
>>> nums = [42, 11]
>>> new = nums[:]
>>> new.append(3)

>>> nums
[42, 11]

>>> new
[42, 11, 3]
```

`new` is a **clone** but not an alias!
Takeaways

• We **cannot change** the value of **immutable** objects such as strings
  • Attempts to copy or to modify them creates a new object
  • No need to worry about aliasing side effects

• We **can change** the value of **mutable** objects such as lists
  • When using the `+=` operator with lists mutates the list!
    • Python secretly calls `.append()`
  • Need to be mindful of **aliasing**; be careful to avoid unintended aliases
  • You can create a "true clone" of a list using slicing or by creating a new list containing the same items (e.g., using a loop or list comprehension)
Advanced:
Aliasing in Nested Lists
Nested Lists: Aliasing Nightmare

- Nested lists create more complicated aliasing side effects
- An assignment to a new variable **creates a new list**

```python
>>> list1 = [1, 2, 3]
>>> list2 = [list1]
```
(Crazy) Aliasing Examples

```python
>>> nums = [23, 19]
>>> words = ["hello", "world"]
>>> mixed = [12, nums, "nice", words]

>>> words += ["sky"]
>>> mixed
```

```python
[12, [23, 19], "nice", "hello", "world", "sky"]
```
(Crazy) Aliasing Examples

```python
>>> nums = [23, 19]
>>> words = ["hello", "world"]
>>> mixed = [12, nums, "nice", words]
```
(Crazy) Aliasing Examples

```python
>>> words += ['sky']
```

```
[23, 19]
['hello', 'world', 'sky']
[12, 'nice', ]
words
mixed
```