CSC110 Lecture 14: Variable Reassignment and Object Mutation

David Liu, Department of Computer Science

*Navigation tip for web slides: press **?** to see keyboard navigation controls.*

Announcements and today’s plan

Assignment 2 done!

Announcements

- Assignment 1 and Term Test 1 grades released
- PythonTA Survey 1 out
- Prep 6 will be released later today
- Assignment 3 will be released today or tomorrow

Story so far: variables and values

So far, we've treated variables and values in a Python program as unchanging.

```
x = 10
```

Let $x = 10$.

But sometimes change is good!

```
sum_so_far = 0

for number in numbers:
    sum_so_far = sum_so_far + number
```

```
>>> david = Person('David', 'Liu', 100, '40 St. George Str
```

```
>>> # Increase David's age by 1
>>> david.age = david.age + 1
```

Today you'll learn to...

1. Explain the behaviour of Python code that uses variable reassignment.
2. Define the term object and explain the three fundamental components of an object.
3. Define the terms object mutation and mutable/immutable data types.
4. Write code involving variable reassignment and object mutation.
5. Define and use augmented assignment statements.
6. Differentiate between variable reassignment and object mutation.

Variable reassignment

Recall the accumulator pattern

```
def my_sum(numbers: list[int]) -> int:
    """Return the sum of the given numbers."""
    sum_so_far = 0

    for number in numbers:
        sum_so_far = sum_so_far + number

    return sum_so_far
```

`sum_so_far = sum_so_far + 100`

vs.

$x = x + 100$

Variable reassignment: assigning a value to a variable that has already been defined.

Any variable can be reassigned to any value, regardless of its current value.

```
x = 10
x = 123456
```

```
y = [1, 2, 3]
y = 'David is cool'
```

Variable “dependencies” and reassignment

```
x = 1
y = x + 2
x = 7
```

Variable reassignment only changes the immediate variable being reassigned, and does not change any other variables or objects, even ones that were defined using the variable being reassigned.

Variable	***Value***
x	~~1~~ **7**
y	3

Augmented assignment

```
sum_so_far = sum_so_far + number
```

Increasing a numerical variable by a given amount is such a common operation that Python (and many other programming languages) provide a short-hand notation for it.

The following is an **augmented assignment statement**:

```
sum_so_far += number
```

We call += an **augmented assignment operator**.

Augmented assignment statement (+=)

```
<variable> += <expression>
```

When the Python interpreter executes this statement, it:

1. Evaluates `<expression>`
2. Adds the value of `<expression>` to the current value of `<variable>`, and reassigns `<variable>` to the result.*

```
<variable> += <expression>
<variable> = <variable> + <expression>
```

*Except when `<variable>` refers to a `list`. Stay tuned!

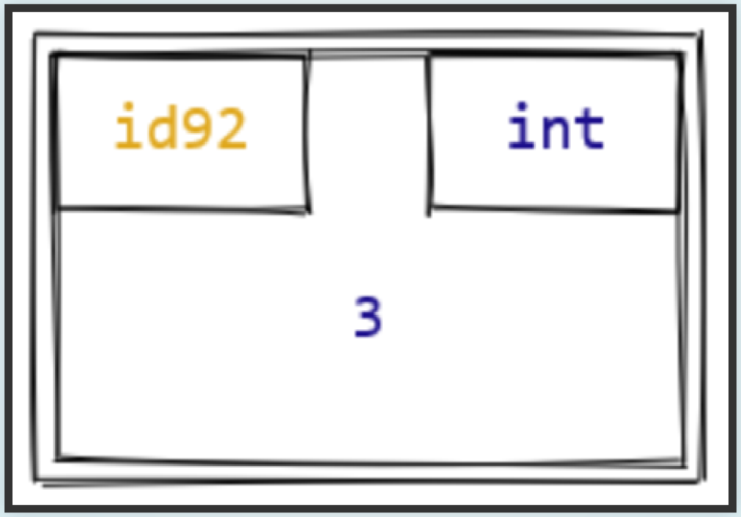
Object mutation

Variable reassignment changes what value a variable refers to.

But, there’s another fundamental way to “change a value” in Python. To understand this, we need to dive deeper into how the Python stores data.

Objects

In Python, every piece of data is stored in an entity called an **object**.



Each object has three fundamental components: id, type, and value.

Object ids

The **id** of an object is a unique identifier that the Python interpreter uses to keep track of the object.

Can view object ids with the built-in `id` function:

```
>>> id('David is cool')
1667609153520
```

Variables refer to objects, and so we can call `id` on them too.

```
>>> s = 'Mario is cool'
>>> id(s)
1667609154288
```

The key idea

An object is like a “box” that the Python interpreter uses to store data.

Once an object is created, its id and type can never change, but (depending on its data type), its value may change.

Mutability

Object mutation is an operation that changes the value of an existing object.

We say a data type is **mutable** when it supports at least one mutating operation, and **immutable** otherwise.

Mutable data types: `list`, `set`, `dict`, data classes

Immutable data types: `int`, `float`, `bool`, `str`

Example: mutating a list

```
>>> numbers = [1, 2, 3]
>>> numbers
[1, 2, 3]
```

```
>>> list.append(numbers, 10)
```

```
>>> numbers
[1, 2, 3, 10]
```

A **mutating function** is a function that mutates one of its arguments. `list.append` is a mutating function (or more precisely, a mutating `list` method).

Mutating methods (demo!)

`list`:

- `list.append`
- `list.insert`
- `list.extend`
- `list.pop`
- `list.sort`

`set`:

- `set.add`
- `set.remove`

`dict`:

- `dict.pop`

Note: These methods mutate an input object, but do not return anything (except `list.pop` and `dict.pop`).

Mutation with an assignment statement

```
<variable> = <expression>
<variable> += <expression>
```

So far, assignment/augmented assignment statements have always had a variable on the left-hand side.

```
<target> = <expression>
<target> += <expression>
```

But we can also use list indexing and dictionary key lookup syntax as the "`<target>`" on the left-hand side of an assignment statement.

This uses assignment to mutate a list/dictionary. **Demo!**

Exercise 1: Reassignment and mutation practice

Mutating data classes

Data class instance attributes can also be the target of an assignment statement.

```
>>> david = Person('David', 'Liu', 100, '40 St. George Str
>>> david.age = 21
>>> david.age
21
```

```
>>> david.age += 100
>>> david.age
121
```

`list` and augmented assignment

```
>>> numbers = [1, 2, 3]
>>> numbers += [4, 5, 6]
>>> numbers
[1, 2, 3, 4, 5, 6]
```

`list` vs. `tuple`

You’ve see that Python has two data types that can represent sequences: `list` and `tuple`.

What’s the difference between them?

`list`s are mutable, but `tuple`s are immutable! (**Demo**)

Because `tuple`s support fewer operations than `list`s, their operations can be implemented more efficiently.

In general, the more specialized a data type, the faster its operations can be implemented.

Accumulating a collection

Comprehension version

```
def squares(nums: list[int]) -> list[int]:
    """Return a list of the squares of the given numbers.

    >>> squares([1, 2, 3])
    [1, 4, 9]
    """
    return [n ** 2 for n in nums]
```

Using variable reassignment

```
def squares(nums: list[int]) -> list[int]:
    """Return a list of the squares of the given numbers."
    squares_so_far = []

    for num in nums:
        squares_so_far = squares_so_far + [num ** 2]

    return squares_so_far
```

Using object mutation

```python
def squares(nums: list[int]) -> list[int]:
    """Return a list of the squares of the given numbers."
    squares_so_far = []

    for num in nums:
        list.append(squares_so_far, num ** 2)

    return squares_so_far
```

```
# Variable reassignment version
for num in nums:
    squares_so_far = squares_so_far + [num ** 2]

# Object mutation version
for num in nums:
    list.append(squares_so_far, num ** 2)
```

Is there a difference? **Demo!**

One advantage of object mutation is that it often faster than creating brand-new objects!

Especially important when using a collection as an accumulator.

Exercise 2: Loops with collection accumulators

Worked example: grouping tabular data

```
def total_marriages_by_centre(data: list[MarriageData]) -> dict[str,
    """Return mapping from civic centre name to the total number of
    marriage licenses ever issued by that centre.
    """
```

Reassignment, mutation, and reasoning about code

```
def my_function(...) -> ...:
    x = 10
    y = [1, 2, 3]

    ...  # Many lines of code
    ...  # Many lines of code
    ...  # Many lines of code
    ...  # Many lines of code

    return x * len(y) + ...
```

What are the values of `x` and `y` in the final return statement? Can we tell?

Tips on using variable reassignment and mutation

- **Do** use reassignment/mutation in loops
- **Do** use mutation for collection accumulators (instead of creating new objects)
- **Do** use mutation when changing part of a larger object
- **Avoid** unnecessary reassignments/mutation
- **Avoid** changing a variable’s value in many different places
- **Avoid** mutating function arguments unless described in the function docstring

Summary

Today you learned to...

1. Explain the behaviour of Python code that uses variable reassignment.
2. Define the term object and explain the three fundamental components of an object.
3. Define the terms object mutation and mutable/immutable data types.
4. Write code involving variable reassignment and object mutation.
5. Define and use augmented assignment statements.
6. Differentiate between variable reassignment and object mutation.

Homework

- Readings:
  - From today: 6.1, 6.2, 6.3
  - For next class: 6.4, 6.5, 6.6, 6.7
  - For the prep: 7.1
- Complete PythonTA Survey 1
- Look for Assignment 3 (to be posted soon)

No meme, just mutant appreciation