

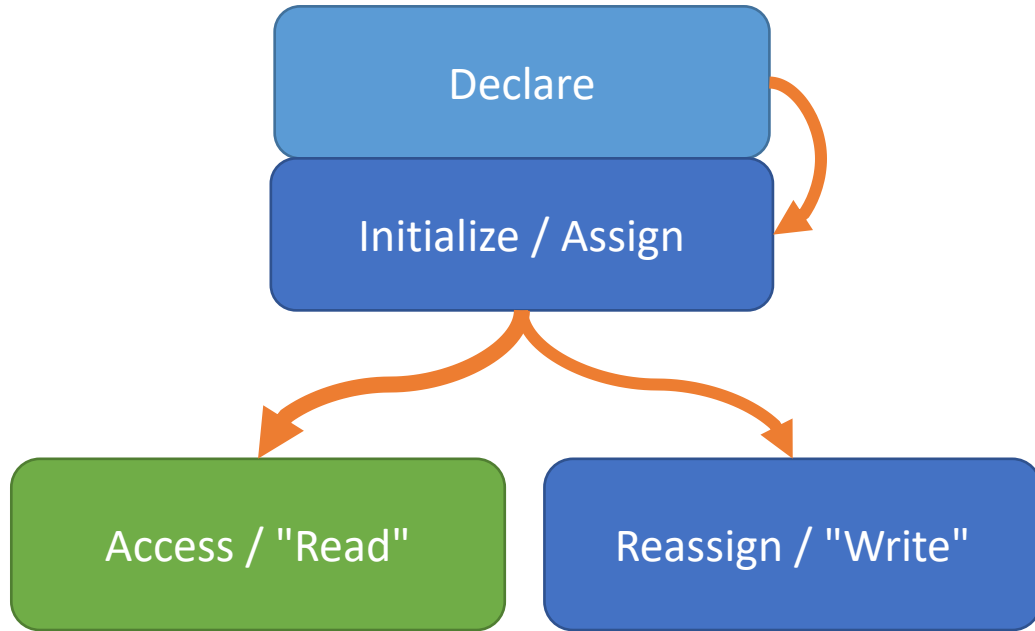
Variables

- **Variables** allow your programs to *store, load, and change* values in memory.
- *Every* variable:
 1. has a **name** and
 2. is bound to a value of a specific **data type**

age
int

20

How to use a variable, generally...



1. **Declare** the variable with name & type
2. **Initialize / Assign** variable its first value
(Steps 1 and 2 can be combined!)

Once 1 and 2 are done, then you can*:

- **Access** the value stored in a variable, or,
- **Reassign** new values to the variable

* There are additional rules governing where you can access and assign a variable from.

Variable Declaration Syntax

- When you **declare** a variable, you are proclaiming...
“henceforth, the identifier <some name> shall be bound to a(n) object of <some type> stored in memory”

age: int

- “the identifier **age** shall refer to an **int** value stored in memory.”
- General form:
[identifier]: [type]
- The type can be: **int, float, str**
(and *many more* types to come!)

Variable Assignment Syntax (1/4)

- The assignment statement **binds** a value to a variable

age = 21

- “age is bound to the value 21”
- General form:
[identifier] = [expression]
- The single equal symbol's name is the **assignment operator**.

Variable Assignment Semantics (2/4)

`age = 20`

The identifier `age` is bound to a space in memory holding the value 20.

Later, if the following statement evaluates:

`age = 21`

The identifier `age` is now bound to a space in memory holding the value 21.

Current Scope - *after age = 20 evaluates*

`age` 20

Current Scope - *after age = 21 evaluates*

`age` 21

Assignment is *not* equality!

Variable Assignment Rules (3/4)

- **A variable's value can change** as the program runs
 - Just assign another value to the same variable!
 - After an assignment statement evaluates, when a subsequent line of code accesses the variable it will have the most recently assigned value.
- **The assignment operator is not commutative!**
 - `[identifier] = [expression] # OK`
 - `[expression] = [identifier] # NOT OK`
 - The variable's name must be on the left of the assignment operator (=) and the value being assigned must be on the right.*
- **You should not refer to a variable until after its name defined and bound!**
 - Try: `print(unbound_variable)`
 - Result: `NameError: name 'unbound_variable' is not defined`
- **For COMP110: expression's type *must match* the variable's declared type**

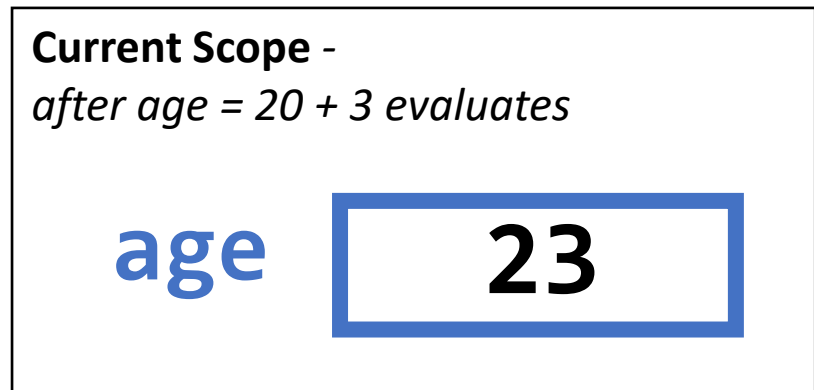
Variable Assignment Rules - Expressions (4/4)

- Notice the *right-hand side (RHS)* of assignment is an **expression!**
[identifier] = [expression]
- Remember! *Every expression evaluates to a single value at runtime.*
- To know *what* value the variable name will be bound to, the expression of an assignment statement must first be evaluated.

- If the following line ran:

age = 20 + 3

1. The computer evaluates the RHS expression
2. The name age is bound to the result of it



Variable Initialization (1 / 2)

- **Initialization** is the *first* time you assign a value to a variable.
 - After initialization a variable is considered *defined* or "*bound*".

- **Always, always, always initialize your variables!**

- You can **declare** *and* **initialize** it in two steps:

```
lucky: int  
lucky = 13
```

- Or, you can combine these steps into a single statement:

```
lucky: int = 13
```


Variable Initialization – Type Inference (2 / 2)

- Notice there is some redundancy in this statement:

```
lucky: int = 13
```

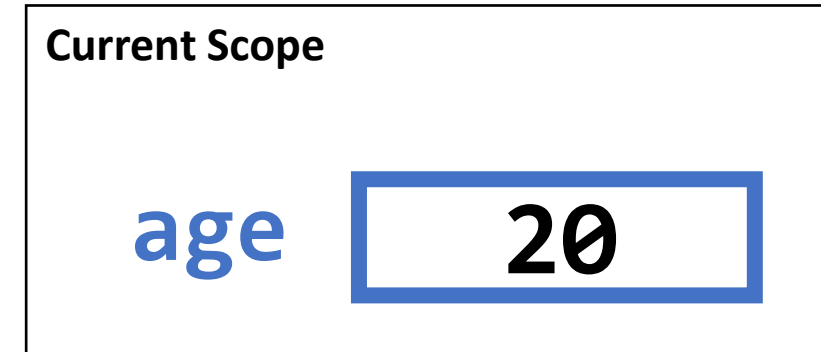
- "Let lucky be an *int* variable that is initially assigned the *int* 13."
- If you combine declaration and initialization, a modern programming language will *infer* the variable's type for you. So, you can write:

```
lucky = 13
```

- You are encouraged to explicitly type your variables in this course, even when there is some redundancy in the declaration. It will help you avoid accidental typing errors!

Variable Access Expression – "Read" (1/2)

- *After* you have declared a variable *and* initialized it...
- You can **access** ("read", "look up") a variable's value in memory **by its name**




age

- "Find the name age and evaluate to its bound value."
- Caution! This is *very different* than: "age"
 - This is a string literal expression!

Variable Access in an Assignment Statement (2/2)

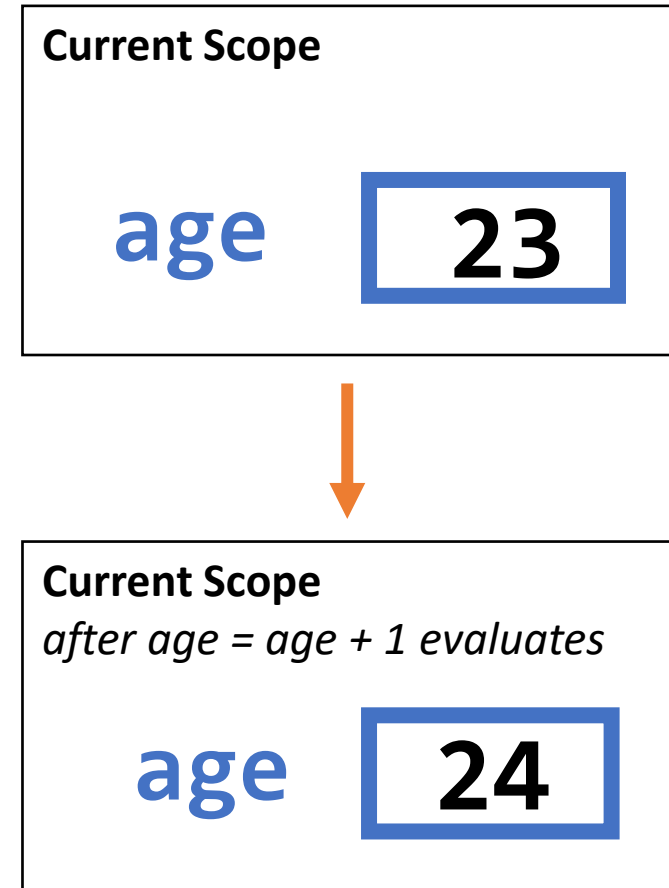
- Consider the following assignment statement:


age = age + 1

“age is assigned the current value of age plus one”

Steps:

1. current value of **age** is accessed ("read")
2. The integer value 1 is added to it
3. **age** is bound to the resulting value in memory



Variable Name & Identifier Rules (1/2)

Variable names are an example of an *identifier*.

Identifiers cannot contain spaces, must begin with a letter or underscore, and contain only letters, numbers, and underscores.

In Python, it is traditional to use **snake_casing** for multiword variable names.

For example, a variable to store "year of birth" would be named:

year_of_birth

Pythonic: Dunderscore Identifiers (1/2)

Python surrounds special identifiers in **double underscores** called *dunderscores*

Example: `__author__`

These are identifiers Python assigns special meaning to. We'll see more!

`__init__`

`__name__`

`__repr__`

`__str__`

This is a Pythonic **Idiom**! Each language has its own idioms for similar purposes.