COMP 110

CL01

An Introduction to Coding

This part of the lecture…

- Little more lecture-y
- A little more vague

Why?

- A gentler introduction
- Want you to get a bigger picture of the little things we're going to talk about later
- **I don't expect you to be able to do all of these things tomorrow… that's what this class is for!**

Computational Thinking

- Strategic thought and problem-solving
- Can help perform a task better, faster, cheaper, etc.
- Examples:
  - Meal prepping
  - Making your class schedule
  - “Life Hacks”

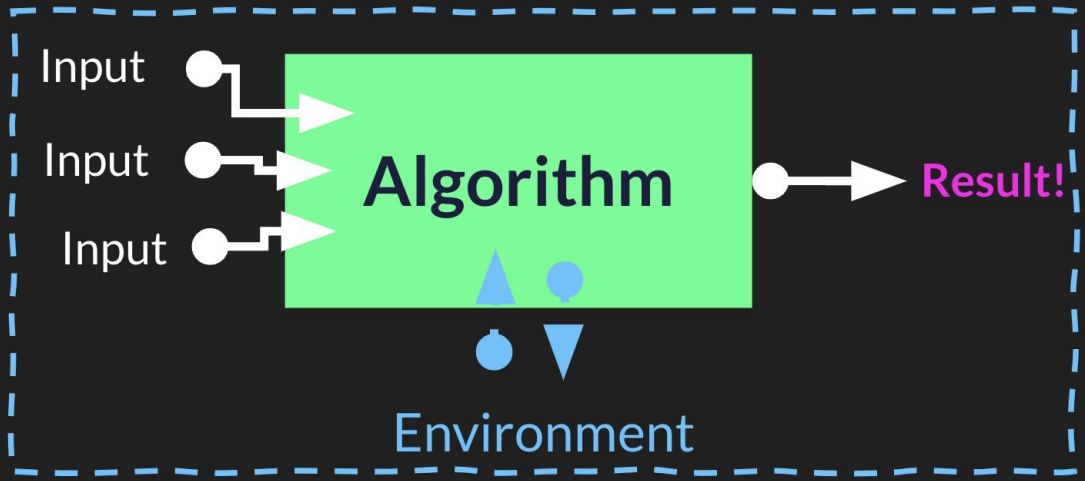
Algorithms

Input is data given to an algorithm

An **algorithm** is a series of steps

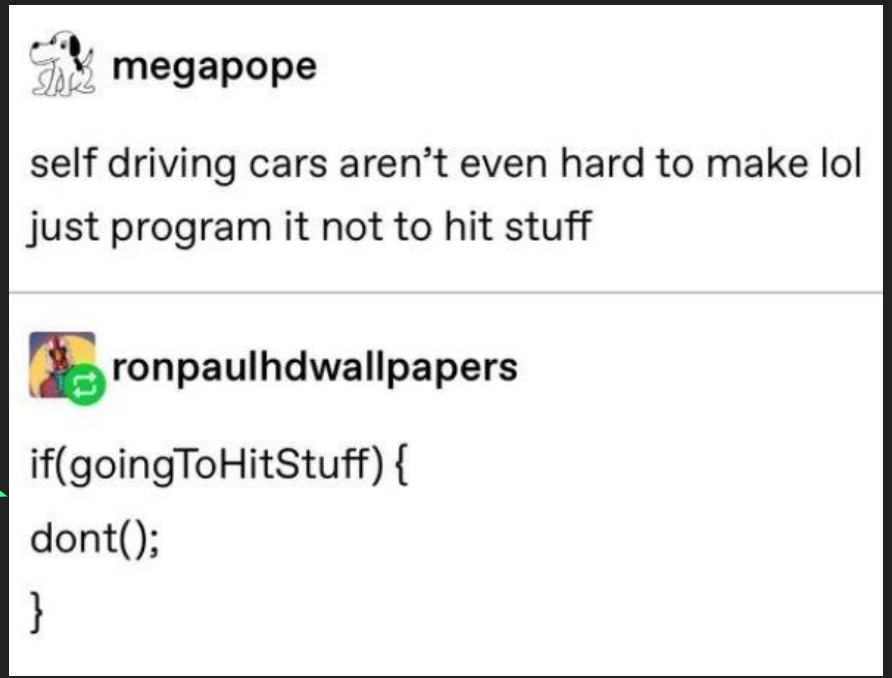
An algorithm **returns** some **result**

An algorithm *may* be influenced by its **environment** and it *may* produce side-effects which influence its environment.



Example: My dissertation

Algorithm



Discussion

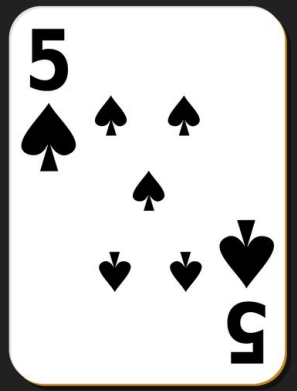
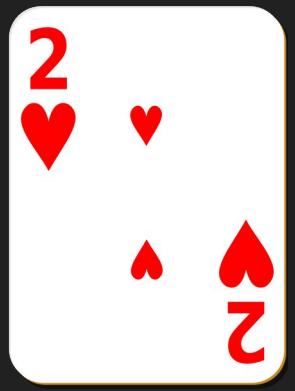
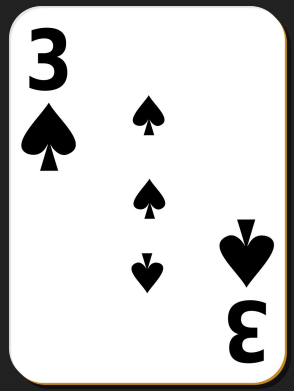
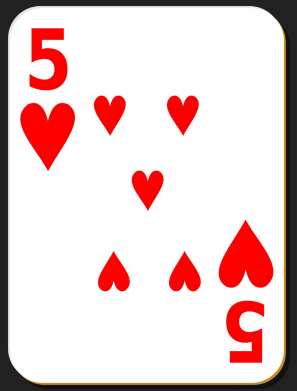
What are examples of computational thinking that you use day to day?

What kind of algorithms do you use to implement these ideas?

What is an algorithm?

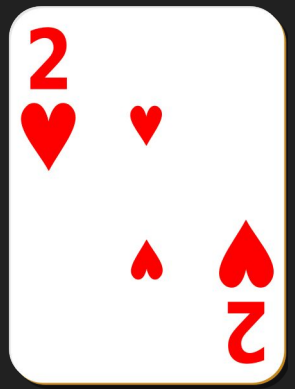
- A set of steps to solve a general problem
- Finite
- Can handle a problem of arbitrary size

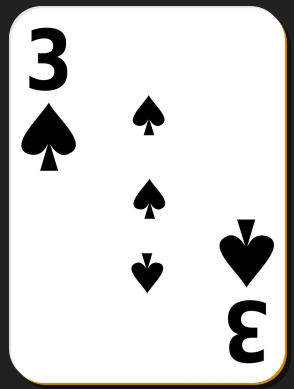
Finding the Lowest Card in a Deck

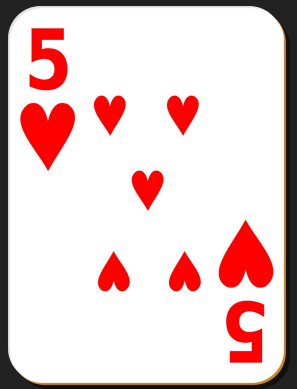





- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

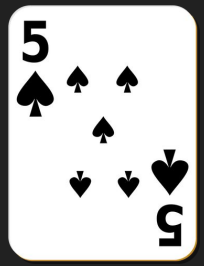
Finding the Lowest Card



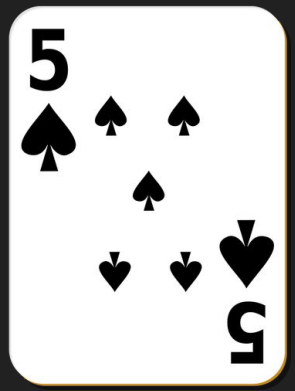


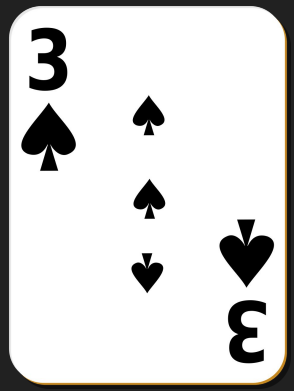


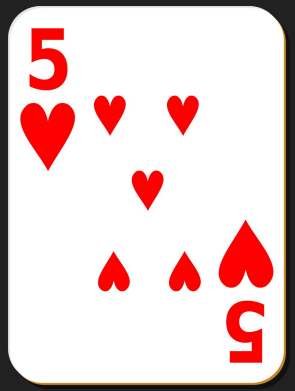
Low card:



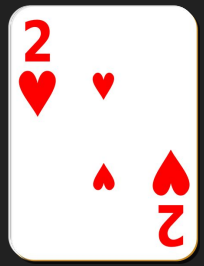
Finding the Lowest Card



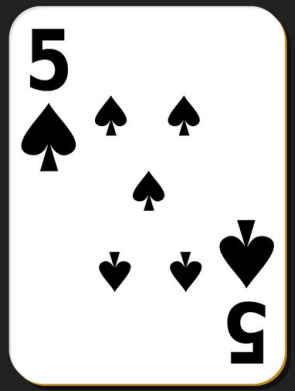


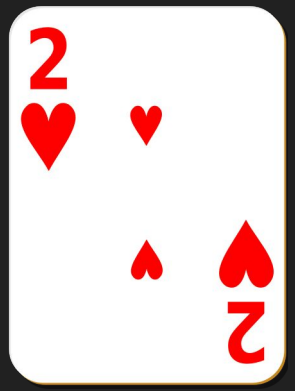


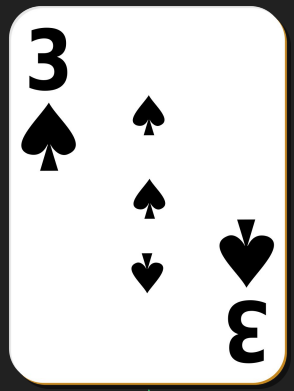
Low card:

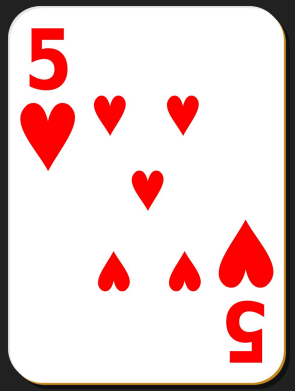


Finding the Lowest Card

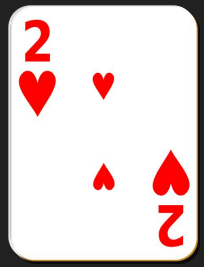




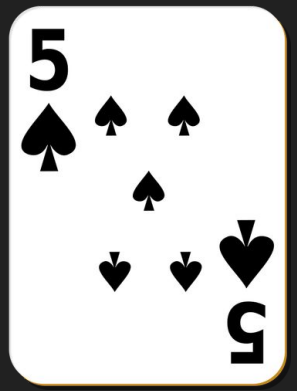


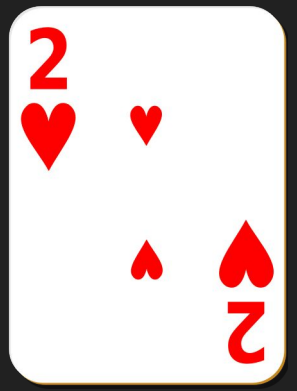


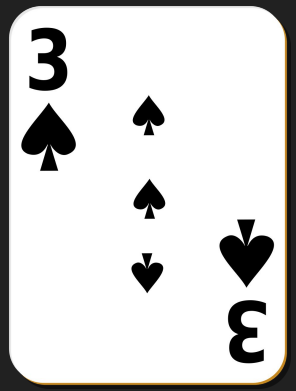
Low card:



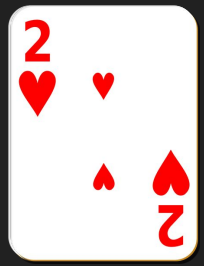
Finding the Lowest Card



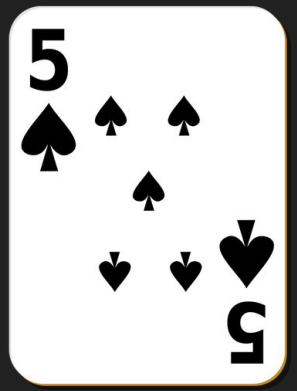


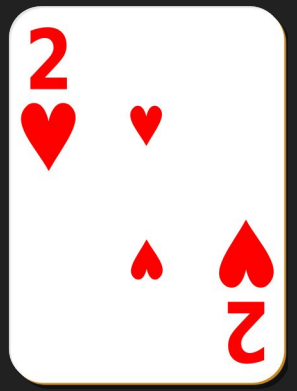


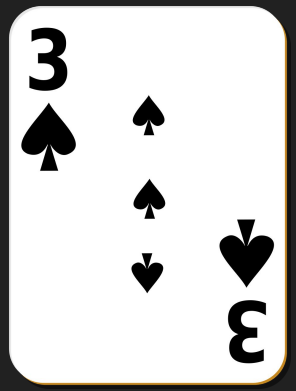
Low card:

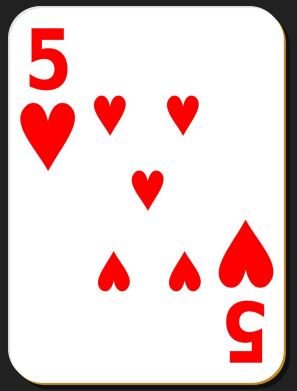


Finding the Lowest Card

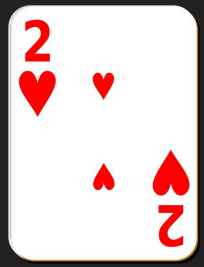




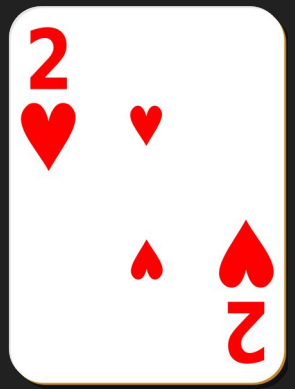


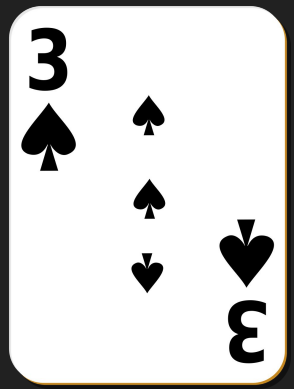


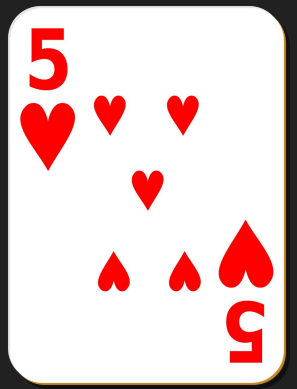
Low card:



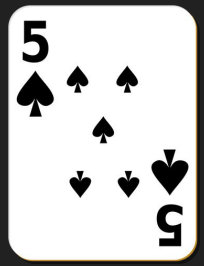
Finding the Lowest Card



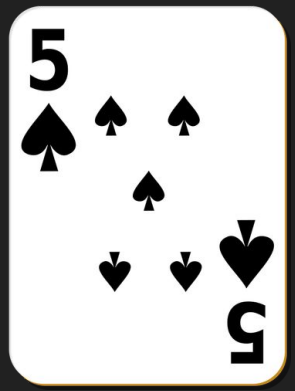


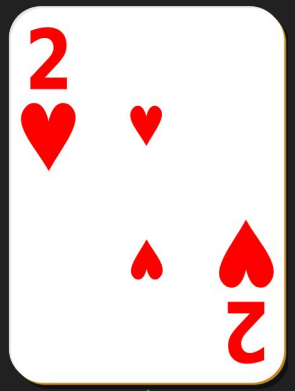


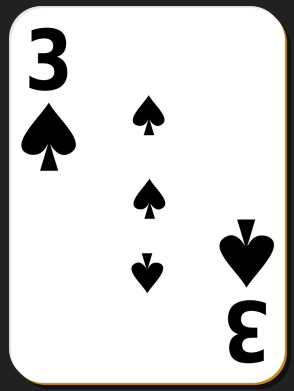
Low card:

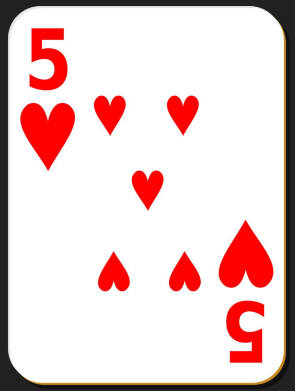


Finding the Lowest Card



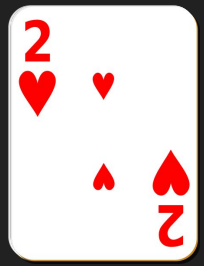




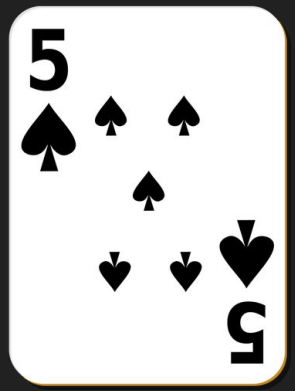


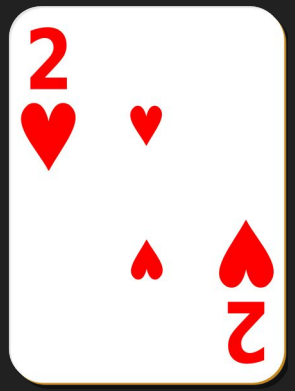
2 < 5? ✓

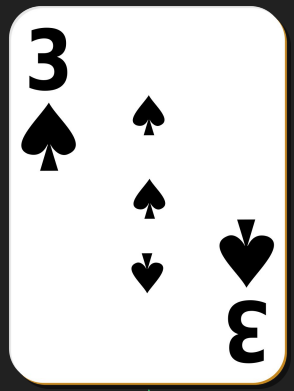
Low card:

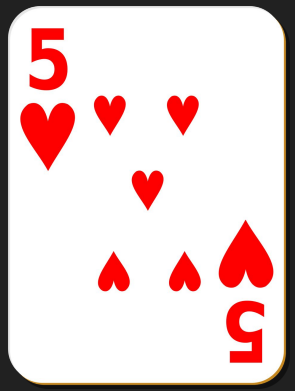


Finding the Lowest Card



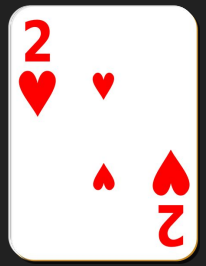




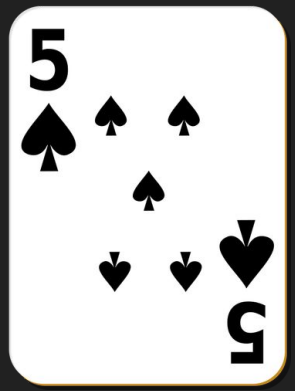


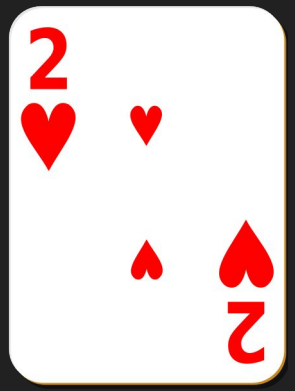
3 < 2?

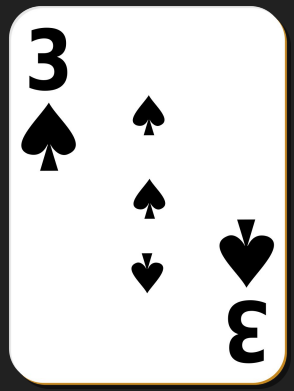
Low card:



Finding the Lowest Card

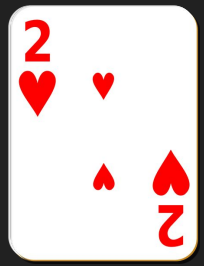




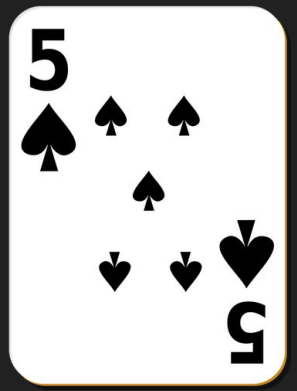


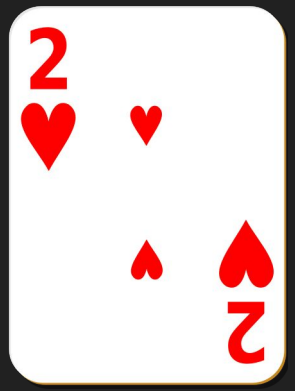
5 < 2? ✗

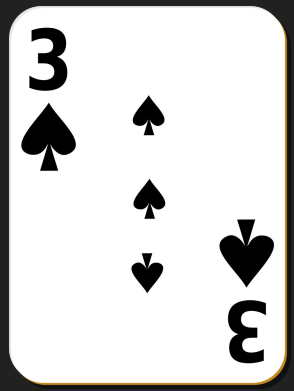
Low card:



Finding the Lowest Card



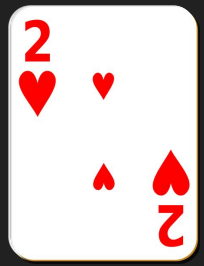




5 < 2?

Relational Operator

Low card:



Pseudocode

Looks like code, but simplified and readable.

Not meant to run on a computer.

Helps you outline what your algorithm is going to look like.

You should be able to expand on your pseudocode to help you write actual code!

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

Pseudocode:

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

Pseudocode:

```
lowest_card = first card in deck
```

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

Pseudocode:

```
lowest_card = first card in deck
```

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

Pseudocode:

```
lowest_card = first card in deck
```

Assignment

(Week 1 concept)

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

Pseudocode:

```
lowest_card = first card in deck

Repeatedly until end of deck:

    if current_card < lowest_card:

        lowest_card = current_card
```

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

Pseudocode:

```
lowest_card = first card in deck
Repeatedly until end of deck:
    if current_card < lowest_card:
        lowest_card = current_card
```

Loop

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

Pseudocode:

```
lowest_card = first card in deck
Repeatedly until end of deck:
    if current_card < lowest_card:
        lowest_card = current_card
```

Loop

(Week 3 concept)

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

Pseudocode:

```
lowest_card = first card in deck

Repeatedly until end of deck:

    if current_card < lowest_card:

        lowest_card = current_card
```

Conditional

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

Pseudocode:

```
lowest_card = first card in deck

Repeatedly until end of deck:

    if current_card < lowest_card:

        lowest_card = current_card
```

Conditional

(Week 1 concept)

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

Pseudocode:

```
lowest_card = first card in deck

Repeatedly until end of deck:

    if current_card < lowest_card:

        lowest_card = current_card
```

Relational Operator

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

Pseudocode:

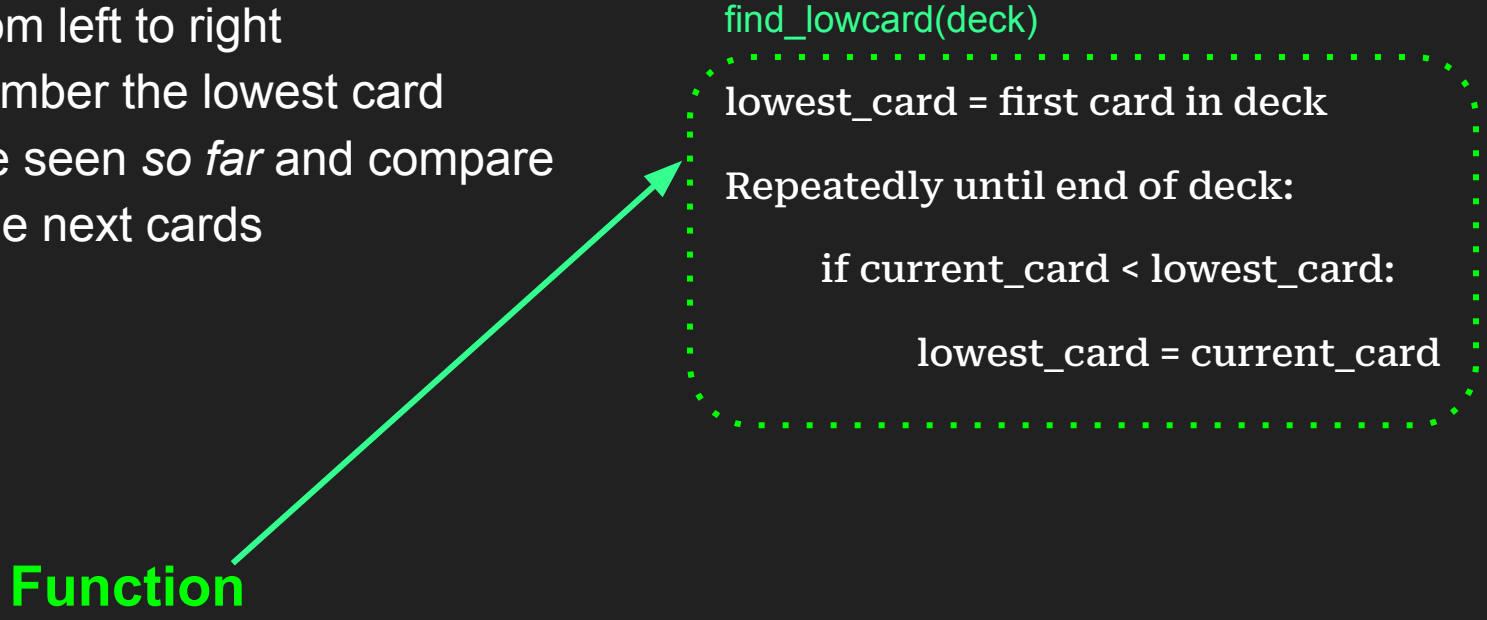
```
lowest_card = first card in deck

Repeatedly until end of deck:

    if current_card < lowest_card:

        lowest_card = current_card
```

Relational Operator

(Week 1 concept)

Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards



Finding the Lowest Card Pseudocode

- Go from left to right
- Remember the lowest card you've seen *so far* and compare it to the next cards

find_lowcard(deck)

```
lowest_card = first card in deck

Repeatedly until end of deck:

        if current_card < lowest_card:

                lowest_card = current_card
```

Function

(Week 4 concept)

Takeaways

- Pseudocode: simple and readable version of algorithm that resembles code
- Assignment Operator: Assigns a variable some value
- Loop Statement: Repeatedly performs an action a fixed number of times
- Relational Operator: Compares two values
- Conditional Statement: A statement that only performs an action under certain conditions
- Function: Generalizes code to work for a generic input

Again, you don't need to know these right now, but I want you to have a point of reference when you do learn them!

Commenting

Commenting

- Comments are text meant to be *read by you*, not interpreted as code by Python!
- To help you (or others) look back at your code and know what you were thinking!
- Single line comment: # my comment here
- Multi-line comment

```
“““
Write multiple things here.
And more here.
”””
```

Objects and Data Types

Objects and Types

An **object** is *typed* unit of data in memory.

The object's **type** classifies it to help the computer know how it should be interpreted and represented.

Example types of data:

- Numerical
- Textual
- Sequences
- Grouping of different types

Numerical Built-In Types

- Integers
  - `int`
  - Zero or non-zero digit followed by zero or more integers (e.g. 100 is an int but 0100 is not)

- Decimals (Or floats)
  - `float`
  - Not the only way to represent decimal numbers, but a very precise way

Textual Built-In Type

- Strings
  - `str`
  - A sequence (or *string*) of characters
  - Can be denoted using “ ”

Indexing

- **Subscription** syntax uses square brackets and allows you to access an item in a sequence
- **Index numbering starts from 0**

Docstrings

- A string written at the top of every file to describe its purpose.
- Denoted with three quotations “““ ”””

Booleans

- `bool`
- Evaluates to `True` or `False`

Check an Object's Type

- type()

Change an Object's Type

- float()
- str()
- int()

Pause to practice:

Please do the LS on Gradescope!

Expressions

Expressions

- Something that *evaluates* at runtime
- Every expression evaluates to a specific typed value
- Examples
  - 1 + 2 * 3
  - 1
  - 1.0 * 2.0
  - “Hello” + “ World!”
  - 1 > 3

Numerical Operators

Operator Name	Symbol
Addition	+
Subtraction/Negation	-
Multiplication	*
Division	/
Exponentiation	**
Remainder “modulo”	%

Addition +

- If numerical objects, add the values together
  - 1 + 1 → 2
  - 1.0 + 2.0 → 3.0
- If strings, concatenate them
  - “Comp” + “110” → “Comp110”
- The result type depends on the operands
  - float + float → float
  - int + int → int
  - float + int → float
  - int + float → float
  - str + str → str

Addition +

- If numerical objects, add the values together
  - 1 + 1 → 2
  - 1.0 + 2.0 → 3.0
- If strings, concatenate them
  - “Comp” + “110” → “Comp110”
- The result type depends on the operands
  - float + float → float
  - int + int → int
  - float + int → float
  - int + float → float
  - str + str → str

Question: What happens when you try to add incompatible types?

Subtraction/Negation -

- Meant strictly for numerical types
  - 3 - 2 → 1
  - 4.0 - 2.0 → 2.0
  - 4.0 - 2 → 2.0
  - - (1 + 1) → -2
- The result type depends on the operands
  - float - float → float
  - int - int → int
  - float - int → float
  - int - float → float

Multiplication *

- If numerical objects, multiply the values
  - 1 * 1 → 1
  - 1.0 * 2.0 → 2.0
- If string and int, repeat the string
  - “Hello” * 3 → “HelloHelloHello”
- The result type depends on the operands
  - float * float → float
  - int * int → int
  - float * int → float
  - int * float → float
  - str * int → str

Division /

- Meant strictly for numerical types
  - 3 / 2 → 1.5
  - 4.0 / 2.0 → 2.0
  - 4 / 2 → 2.0
- Division results in a float
  - float / float → float
  - int / int → float
  - float / int → float
  - int / float → float

Exponentiation **

- Meant strictly for numerical types
  - 2 ** 2 → 4
  - 2.0 ** 2.0 → 4.0
- The result type depends on the operands
  - float ** float → float
  - int ** int → int
  - float ** int → float
  - int ** float → float

Remainder “modulo”

- Calculates the *remainder* when you divide two numbers
- Meant strictly for numerical types
  - 5 % 2 → 1
  - 6 % 3 → 0
- The result type depends on the operands
  - int % int → int
  - float % float → float
  - float % int → float
  - int % float → float
- Note:
  - If x is even, x % 2 → 0
  - If x is odd, x % 2 → 1

Order Of Operations

- P ()
- E **
- MD * / %
- AS + -
- Tie? Evaluate *Left to Right*

Relational Operators

Operator Name	Symbol
Equal?	==
Less than?	<
Greater than?	>
Less than or equal to? (At most)	<=
Greater than or equal to? (At least)	>=
Not equal?	!=

Relational Operators

- Always result in a bool (True or False)
- Equals (==) and Not Equal (!=)
  - Can be used for all primitive types we've learned so far! (bool, int, float, str)
- Every other type
  - Just use on floats and ints
  - (Can *technically* use on all primitive types)

Practice! Simplify and Type

- 2 + 4 / 2 * 2

- 220 >= int((“1” + “1” + “0”) * 2)

Simplify: 2 + 4 / 2 * 2

(Reminder: P E M D A S)

Simplify: 2 + 4 / 2 * 2

What type is 2 + 4 / 2 * 2?

Simplify:

220 >= int((“1” + “1” + “0”) * 2)

Mods Practice! Simplify

- 7 % 2
- 8 % 4
- 7 % 4
- Any even number % 2
- Any odd number % 2

Pause to practice:

Please do the LS on Gradescope!

Variables

Variables

Declaration of a variable

<name>: <type> = <value>

students: int = 300

message: str = “Howdy!”

Update a variable

<name> = <new value>

students = 325

message = “See ya!”

User Input

User input

- input() function: prompts the user for input and returns the response
- Example

```
your_name: str = input(“What is your name?”)
```

Will store the user’s response as the variable your_name.

Pause to practice:

Please do the LS on Gradescope!