

# WELCOME!

(download slides and .py files from  
the class site to follow along)

6.100L Lecture 1

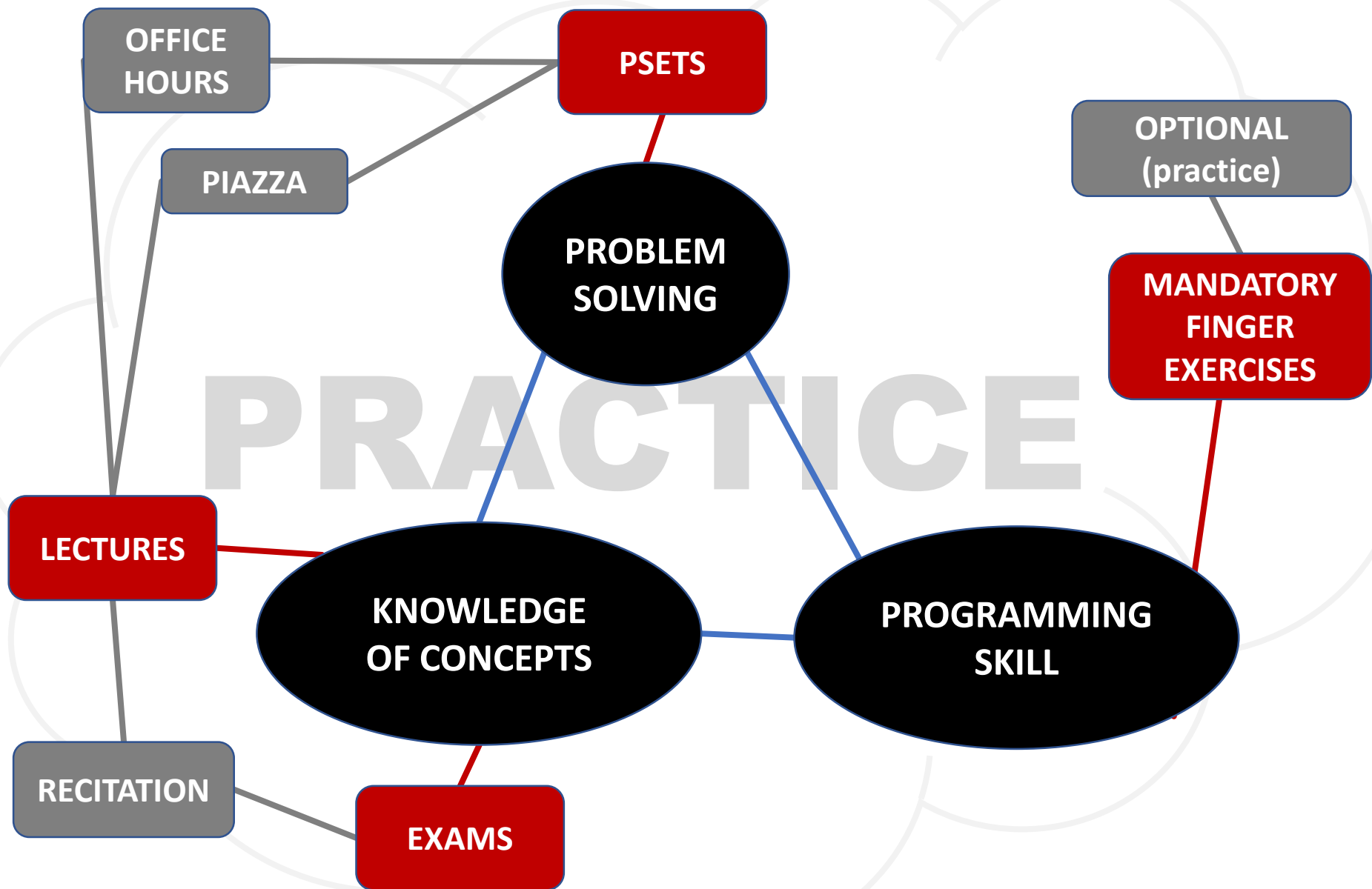
Ana Bell

# TODAY

- Course info
- What is computation
- Python basics
  - Mathematical operations
  - Python variables and types
- NOTE: **slides and code files up before each lecture**
  - Highly encourage you to download them before class
  - Take notes and run code files when I do
  - Do the in-class “You try it” breaks
  - Class will not be recorded
  - Class will be live-Zoomed for those sick/quarantine

# WHY COME TO CLASS?

- You get out of this course what you put into it
- Lectures
  - **Intuition** for concept
  - **Teach** you the concept
  - **Ask** me questions!
  - **Examples** of concept
  - Opportunity to  
**practice practice practice**
  - Repeat



# TOPICS

- Solving problems using **computation**
- Python **programming language**
- Organizing **modular programs**
- Some simple but important **algorithms**
- Algorithmic **complexity**

LET'S GOOOOO!

# TYPES of KNOWLEDGE

- **Declarative knowledge** is **statements of fact**
- **Imperative knowledge** is a **recipe** or “how-to”
- Programming is about writing recipes to generate facts

# NUMERICAL EXAMPLE

- Square root of a number  $x$  is  $y$  such that  $y * y = x$
- Start with a **guess**,  $g$ 
  - 1) If  $g * g$  is **close enough** to  $x$ , stop and say  $g$  is the answer
  - 2) Otherwise make a **new guess** by averaging  $g$  and  $x/g$
  - 3) Using the new guess, **repeat** process until close enough
- Let's try it for  $x = 16$  and an initial guess of 3

$g$	$g * g$	$x / g$	$(g + x / g) / 2$
3	9	$16 / 3$	4.17

# NUMERICAL EXAMPLE

- Square root of a number  $x$  is  $y$  such that  $y * y = x$
- Start with a **guess**,  $g$ 
  - 1) If  $g * g$  is **close enough** to  $x$ , stop and say  $g$  is the answer
  - 2) Otherwise make a **new guess** by averaging  $g$  and  $x/g$
  - 3) Using the new guess, **repeat** process until close enough
- Let's try it for  $x = 16$  and an initial guess of 3

$g$	$g * g$	$x/g$	$(g + x/g) / 2$
3	9	$16/3$	4.17
4.17	17.36	3.837	4.0035

# NUMERICAL EXAMPLE

- Square root of a number  $x$  is  $y$  such that  $y * y = x$
- Start with a **guess**,  $g$ 
  - 1) If  $g * g$  is **close enough** to  $x$ , stop and say  $g$  is the answer
  - 2) Otherwise make a **new guess** by averaging  $g$  and  $x/g$
  - 3) Using the new guess, **repeat** process until close enough
- Let's try it for  $x = 16$  and an initial guess of 3

$g$	$g * g$	$x / g$	$(g + x / g) / 2$
3	9	$16 / 3$	4.17
4.17	17.36	3.837	4.0035
4.0035	16.0277	3.997	4.000002

# WE HAVE an ALGORITHM

- 1) Sequence of simple **steps**
- 2) **Flow of control** process that specifies when each step is executed
- 3) A means of determining **when to stop**

# ALGORITHMS are RECIPES / RECIPES are ALGORITHMS

- Bake cake from a box
  - 1) Mix dry ingredients
  - 2) Add eggs and milk
  - 3) Pour mixture in a pan
  - 4) Bake at 350F for 5 minutes
  - 5) Stick a toothpick in the cake
    - 6a) If toothpick does not come out clean, repeat step 4 and 5
    - 6b) Otherwise, take pan out of the oven
  - 7) Eat

# COMPUTERS are MACHINES that EXECUTE ALGORITHMS

- Two things computers do:
  - Performs simple **operations**  
100s of billions per second!
  - **Remembers** results  
100s of gigabytes of storage!
- What kinds of calculations?
  - **Built-in** to the machine, e.g., +
  - Ones that **you define** as the programmer
- The BIG IDEA here?

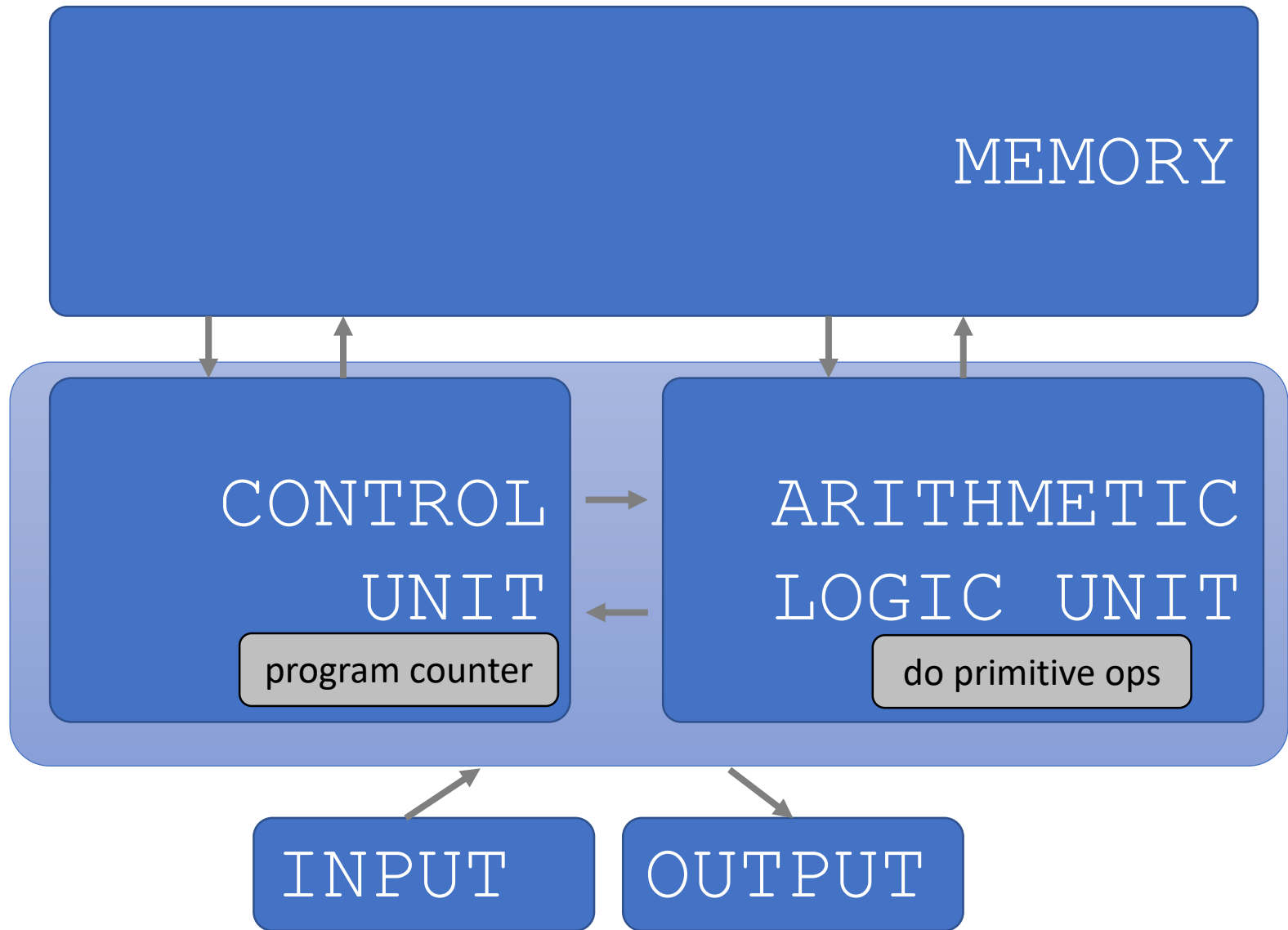
A COMPUTER WILL ONLY DO  
WHAT YOU TELL IT TO DO

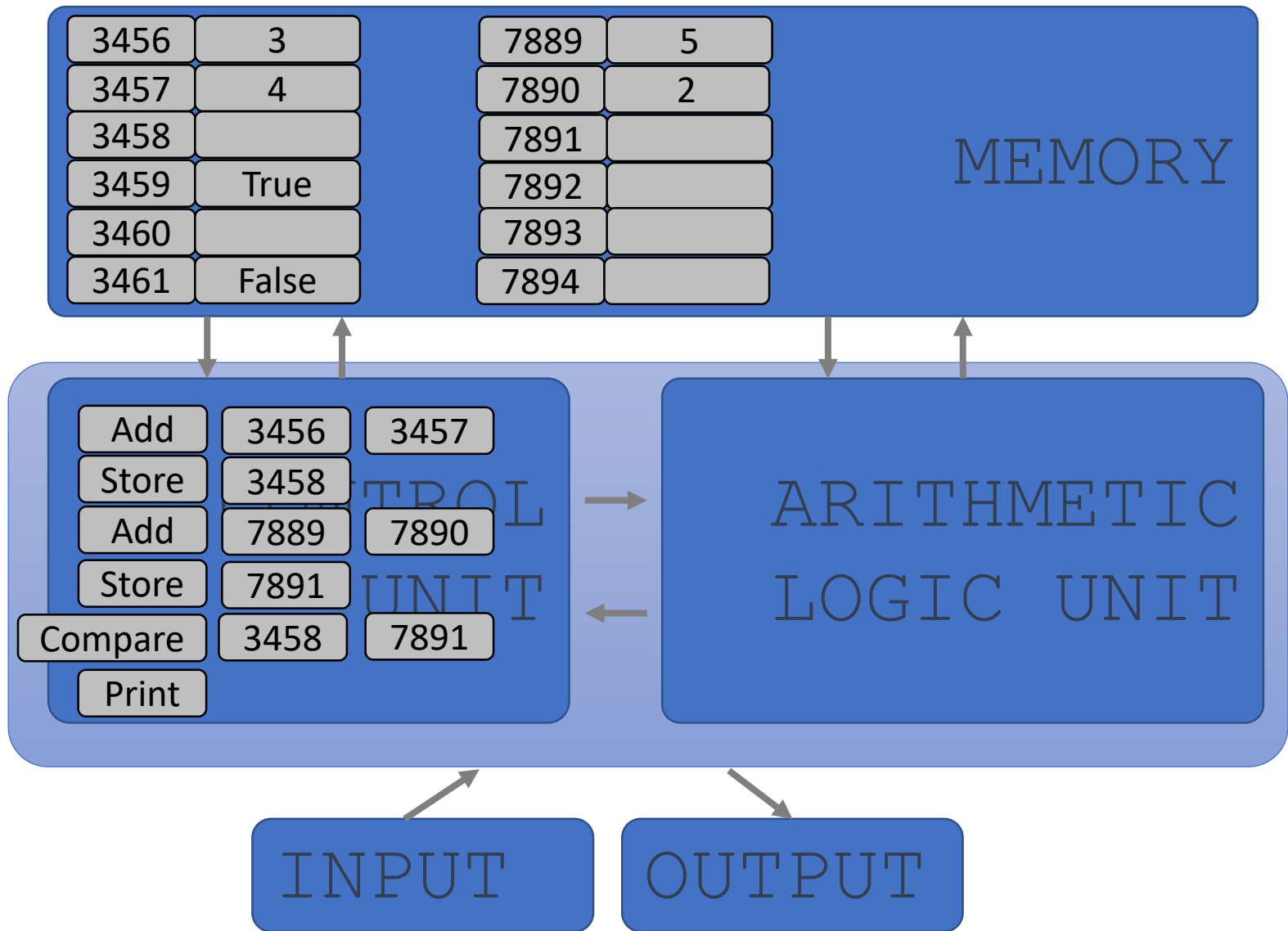
# COMPUTERS are MACHINES that EXECUTE ALGORITHMS

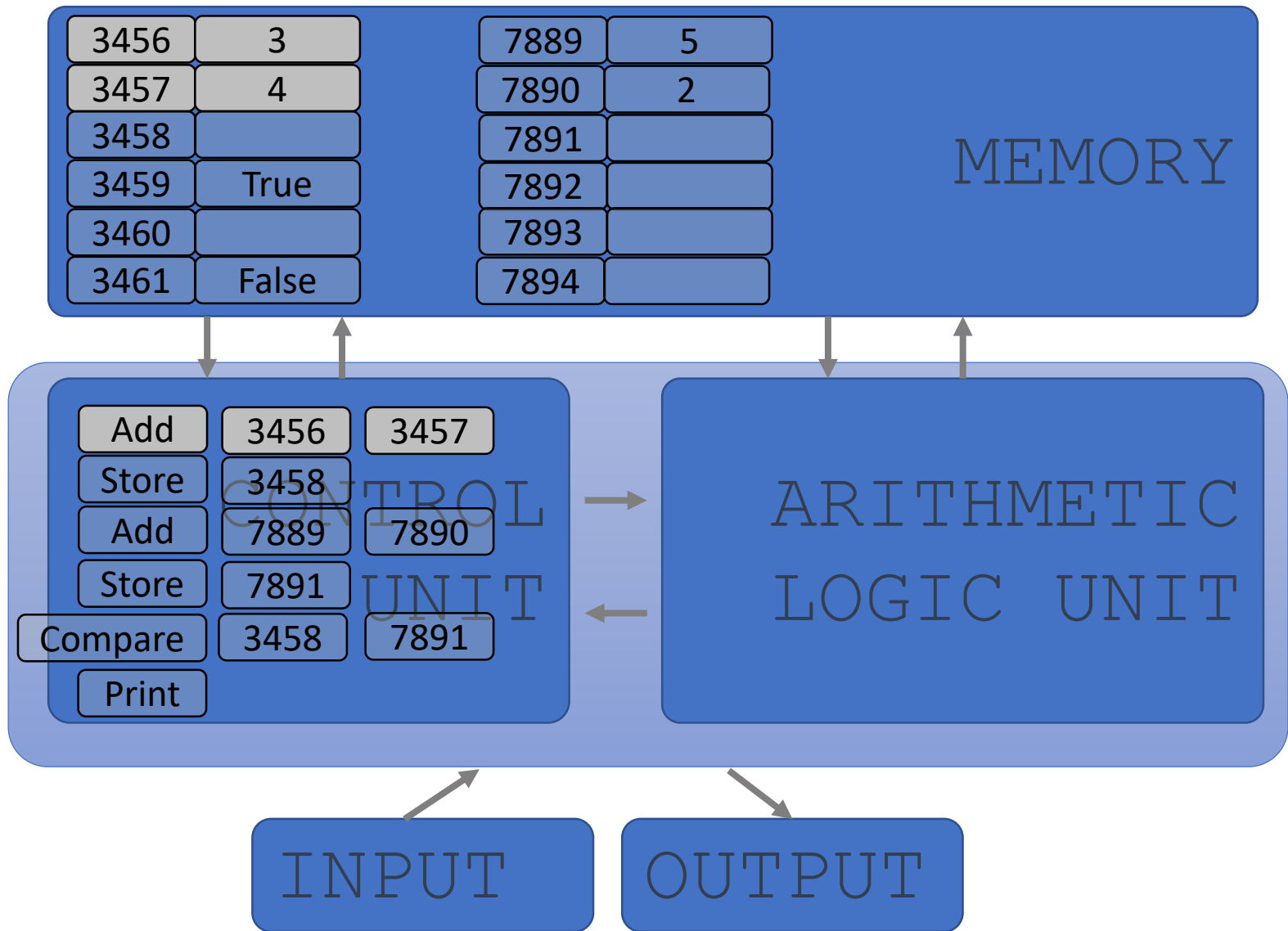
- **Fixed program** computer
  - Fixed set of algorithms
  - What we had until 1940's
- **Stored program** computer
  - Machine stores and executes instructions
- **Key insight:** Programs are no different from other kinds of data

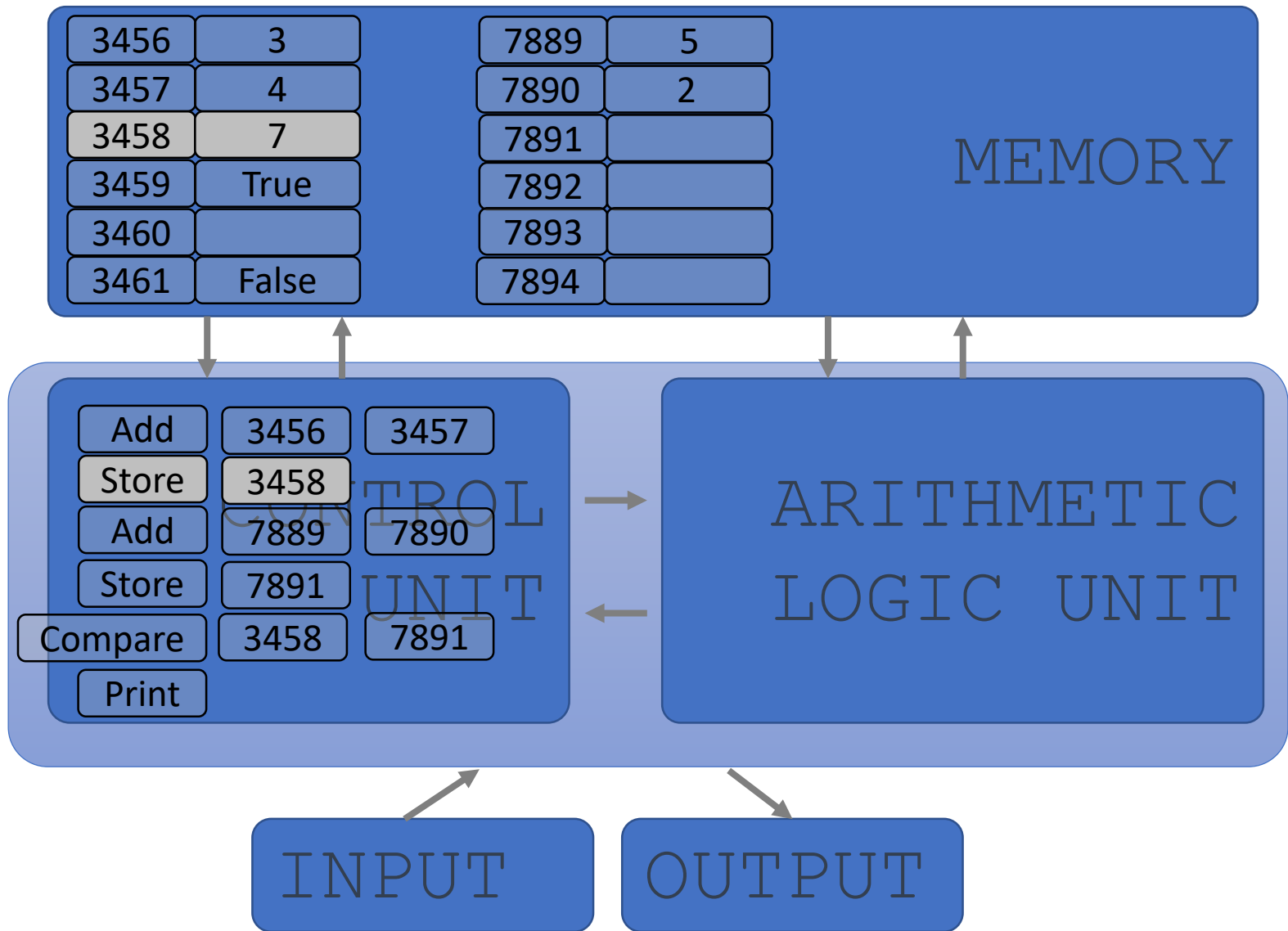
# STORED PROGRAM COMPUTER

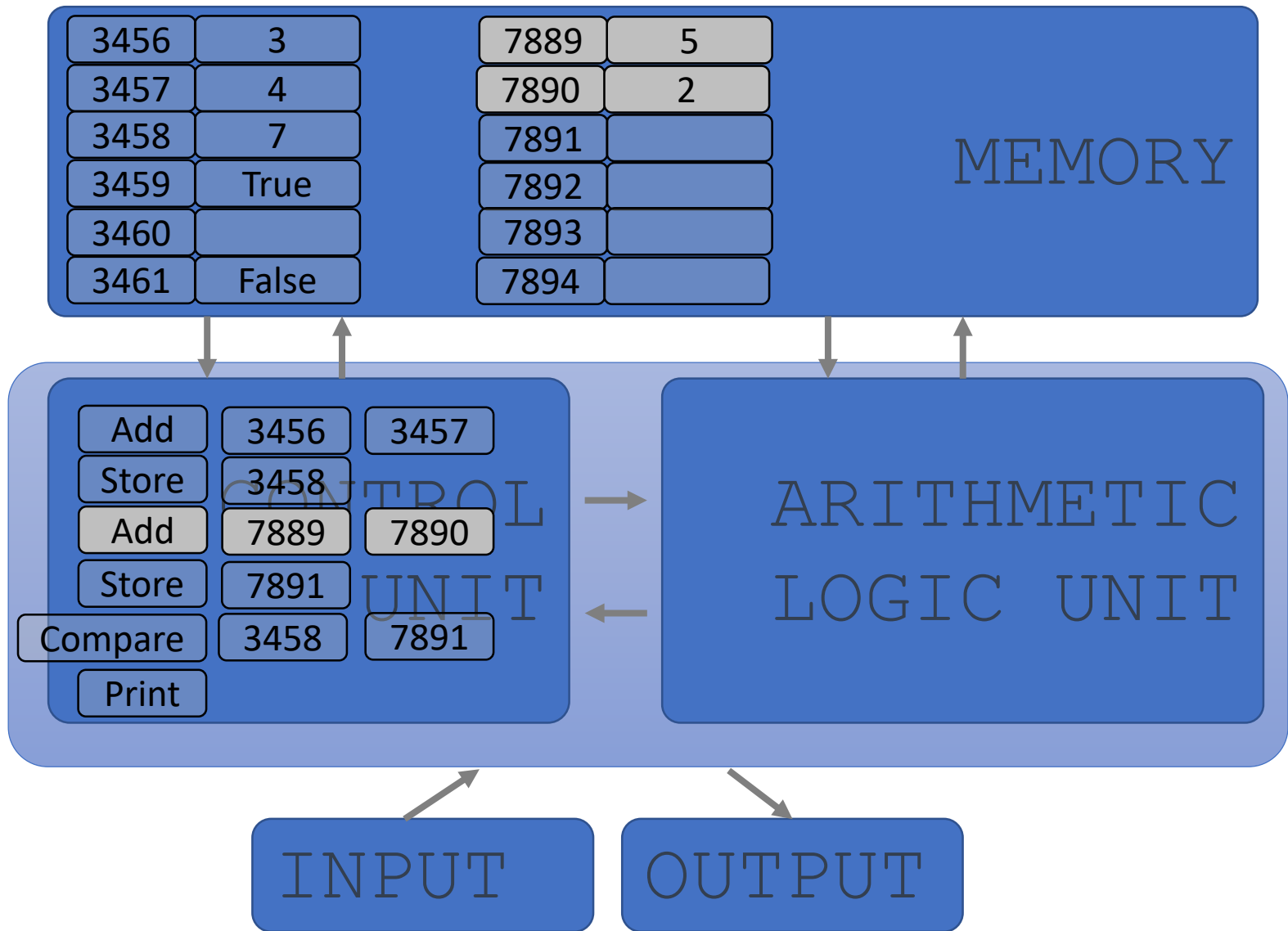
- Sequence of **instructions stored** inside computer
  - Built from predefined set of primitive instructions
    - 1) Arithmetic and logical
    - 2) Simple tests
    - 3) Moving data
- Special program (interpreter) **executes each instruction in order**
  - Use tests to change flow of control through sequence
  - Stops when it runs out of instructions or executes a halt instruction

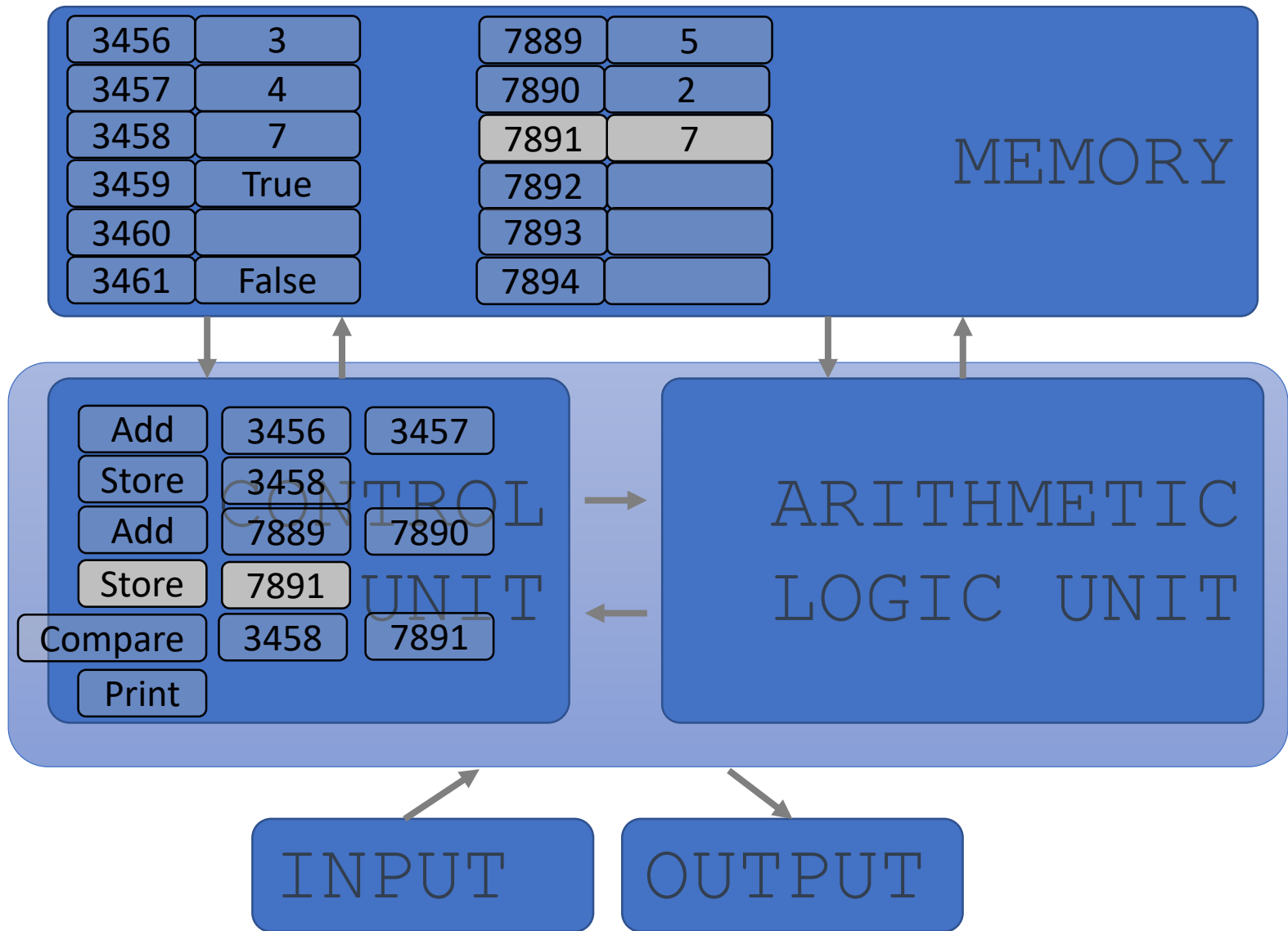


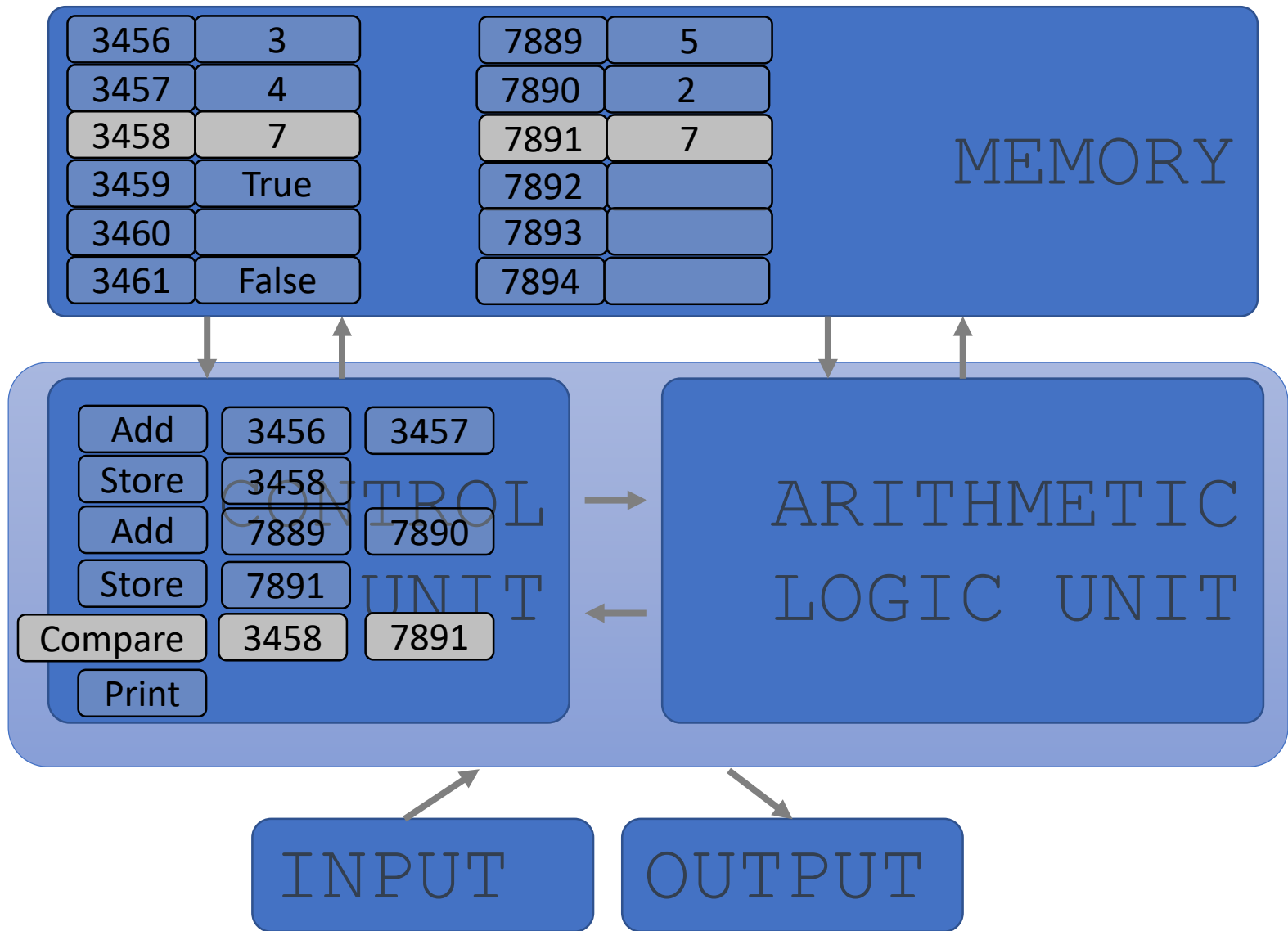


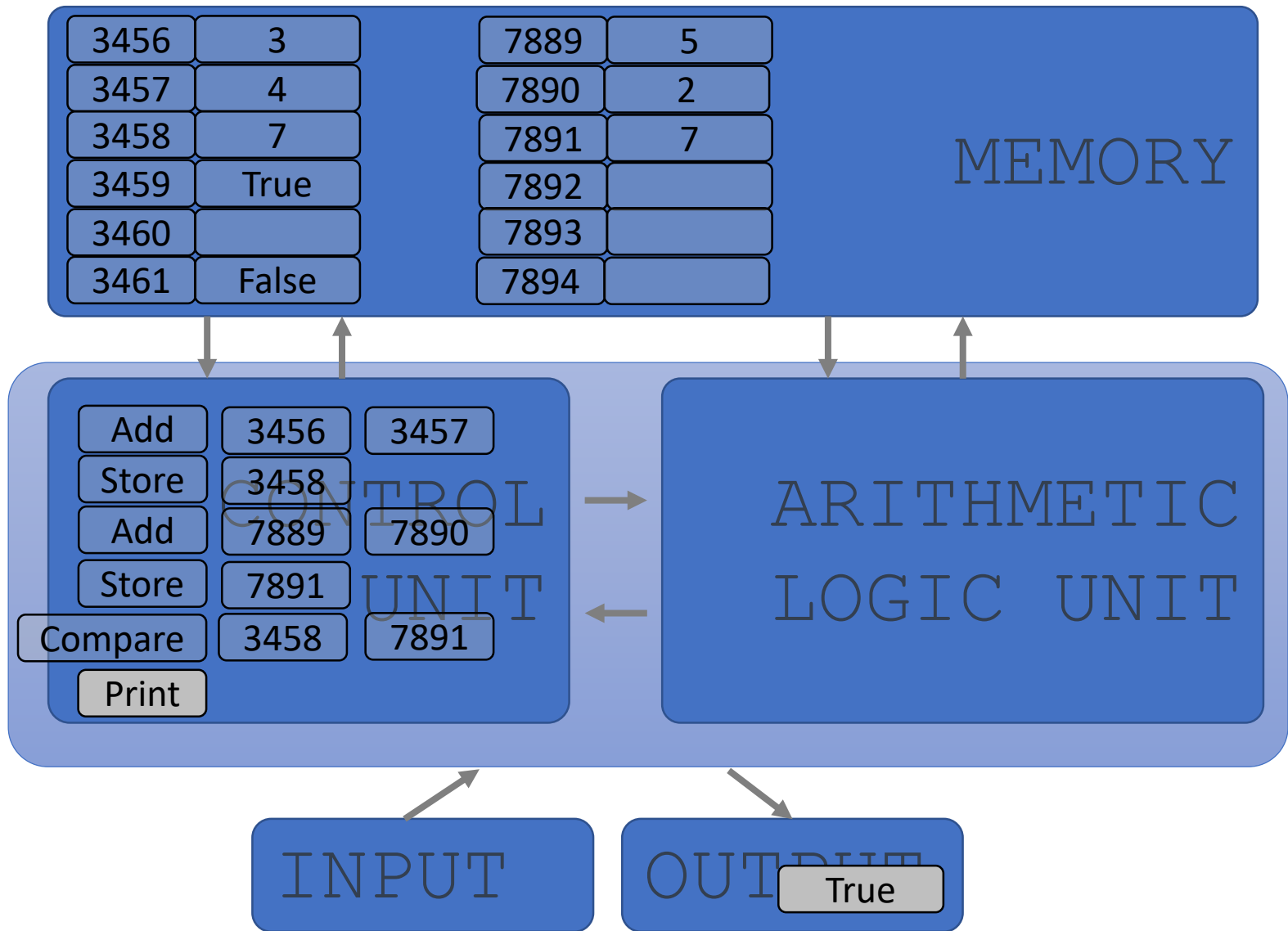






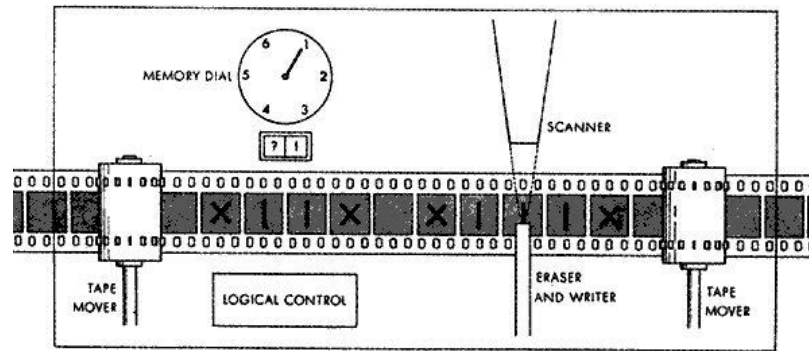






# BASIC PRIMITIVES

- Turing showed that you can **compute anything** with a very simple machine with only 6 primitives: left, right, print, scan, erase, no op



© source unknown. All rights reserved. This content is excluded from our Creative Commons license. For more information, see <https://ocw.mit.edu/help/faq-fair-use/>

- Real programming languages have
  - More convenient set of primitives
  - Ways to combine primitives to **create new primitives**
- Anything computable in one language is computable in any other programming language

# ASPECTS of LANGUAGES

- **Primitive constructs**

- English: words
- Programming language: numbers, strings, simple operators

# ASPECTS of LANGUAGES

## ■ Syntax

- English: `"cat dog boy"` → not syntactically valid  
          `"cat hugs boy"` → syntactically valid
- Programming language: `"hi"5` → not syntactically valid  
                              `"hi"*5` → syntactically valid

# ASPECTS of LANGUAGES

- **Static semantics:** which syntactically valid strings have meaning
  - English: "I are hungry" → syntactically valid  
but static semantic error
  - PL: "hi"+5 → syntactically valid  
but static semantic error

# ASPECTS of LANGUAGES

- **Semantics**: the meaning associated with a syntactically correct string of symbols with no static semantic errors
- English: can have many meanings "The chicken is ready to eat."
- Programs have only one meaning
- **But the meaning may not be what programmer intended**

# WHERE THINGS GO WRONG

- **Syntactic errors**

- Common and easily caught

- **Static semantic errors**

- Some languages check for these before running program
  - Can cause unpredictable behavior

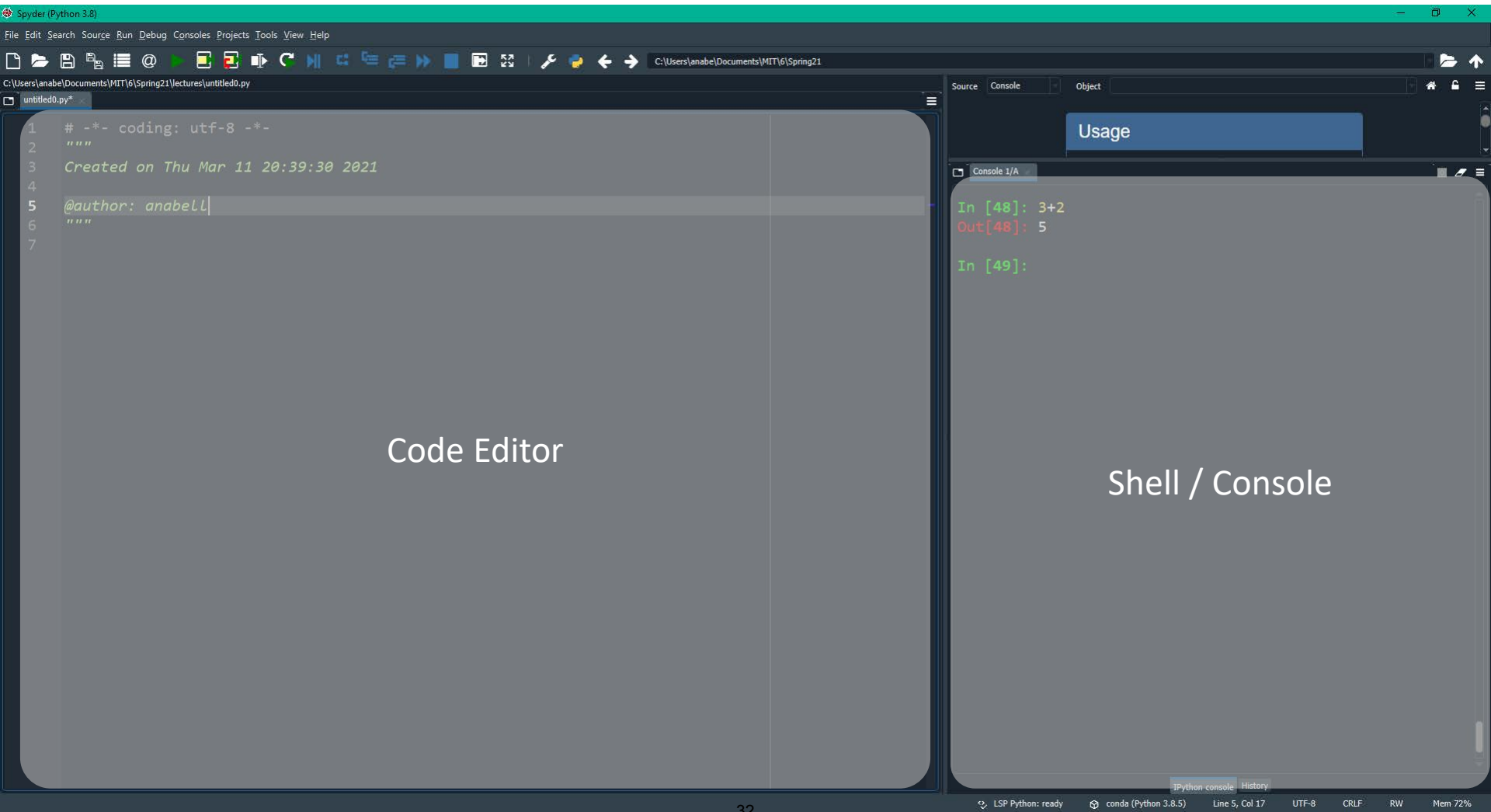
- No linguistic errors, but **different meaning than what programmer intended**

- Program crashes, stops running
  - Program runs forever
  - Program gives an answer, but it's wrong!

# PYTHON PROGRAMS

- A **program** is a sequence of definitions and commands
  - Definitions **evaluated**
  - Commands **executed** by Python interpreter in a shell
- **Commands** (statements) instruct interpreter to do something
- Can be typed directly in a **shell** or stored in a **file** that is read into the shell and evaluated
  - Problem Set 0 will introduce you to these in Anaconda

# PROGRAMMING ENVIRONMENT: ANACONDA



# OBJECTS

- Programs manipulate **data objects**
- Objects have a **type** that defines the kinds of things programs can do to them
  - 30
    - Is a number
    - We can add/sub/mult/div/exp/etc
  - 'Ana'
    - Is a sequence of characters (aka a string)
    - We can grab substrings, but we can't divide it by a number

# OBJECTS

- **Scalar** (cannot be subdivided)
  - Numbers: 8.3, 2
  - Truth value: True, False
- **Non-scalar** (have internal structure that can be accessed)
  - Lists
  - Dictionaries
  - Sequence of characters: "abc"

# SCALAR OBJECTS

- `int` – represent **integers**, ex. 5, -100
- `float` – represent **real numbers**, ex. 3.27, 2.0
- `bool` – represent **Boolean** values `True` and `False`
- `NoneType` – **special** and has one value, `None`
- Can use `type()` to see the type of an object

```
>>> type(5)
```

```
int
```

```
>>> type(3.0)
```

```
float
```

*what you write into the  
Python shell*

*what shows after  
hitting enter*

# int

0, 1, 2, ...  
300, 301 ...  
-1, -2, -3, ...  
-400, -401, ...

# float

0.0, ..., 0.21, ...  
1.0, ..., 3.14, ...  
-1.22, ..., -500.0 , ...

# bool

True  
False

# NoneType

None

# YOU TRY IT!

- In your console, find the type of:
  - 1234
  - 8.99
  - 9.0
  - True
  - False

# TYPE CONVERSIONS (CASTING)

- Can **convert object of one type to another**
  - `float(3)` casts the int 3 to float 3.0
  - `int(3.9)` casts (note the truncation!) the float 3.9 to int 3
- Some operations perform implicit casts
  - `round(3.9)` returns the int 4

# YOU TRY IT!

- In your console, find the type of:
  - `float(123)`
  - `round(7.9)`
  - `float(round(7.2))`
  - `int(7.2)`
  - `int(7.9)`

# EXPRESSIONS

- **Combine objects and operators** to form expressions
  - $3+2$
  - $5/3$
- An expression has a **value**, which has a type
  - $3+2$  has value 5 and type int
  - $5/3$  has value 1.666667 and type float
- Python evaluates expressions and stores the value. It doesn't store expressions!
- Syntax for a simple expression  
`<object> <operator> <object>`

# BIG IDEA

Replace complex  
expressions by ONE value

Work systematically to evaluate the expression.

# EXAMPLES

- `>>> 3+2`

- `5`

- `>>> (4+2) * 6 - 1`

- `35`

- `>>> type((4+2) * 6 - 1)`

- `int`

- `>>> float((4+2) * 6 - 1)`

- `35.0`

*Do computations left to right – like in math!*





*Do computations inside parens first, left to right*

*Take care about what operations you are doing*

# YOU TRY IT!

- In your console, find the values of the following expressions:
  - `(13-4) / (12*12)`
  - `type(4*3)`
  - `type(4.0*3)`
  - `int(1/2)`

# OPERATORS on `int` and `float`

- $i + j \rightarrow$  the **sum** 
  - $i - j \rightarrow$  the **difference** 
  - $i * j \rightarrow$  the **product** 
  - $i / j \rightarrow$  **division**  result is always a float
- if both are ints, result is int  
if either or both are floats, result is float
- $i // j \rightarrow$  **floor division** What is type of output?
  - $i \% j \rightarrow$  the **remainder** when  $i$  is divided by  $j$
  - $i ** j \rightarrow$   $i$  to the **power** of  $j$

# SIMPLE OPERATIONS

- Parentheses tell Python to do these operations first
  - Like math!
- **Operator precedence** without parentheses

\* \*

\* / %      executed left to right, as appear in expression

+ -      executed left to right, as appear in expression

SO MANY OBJECTS, what to do with them?!

a = 2      temp = 100.4  
b = -0.3      go = True  
x = 123      flag = False  
small = 0.001      n = 17

# VARIABLES

- Computer science variables are **different** than math variables

- Math variables**

- Abstract
- Can **represent many values**

$$a + 2 = b - 1$$

$$x * x = y$$

*x represents all  
square roots*

- CS variables**

- Is bound to **one single value** at a given time
- Can be bound to an expression  
(but expressions evaluate to one value!)

$$a = b + 1$$

$$m = 10$$

$$F = m * 9.98$$

*one variable*

*one value*

# BINDING VARIABLES to VALUES

- In CS, the equal sign is an **assignment**
  - One value to one variable name
  - Equal sign is **not equality**, not “solve for x”
- An assignment binds a value to a name

*variable* pi = 355/113 *value*

- **Step 1:** Compute the value on the **right hand side** (the VALUE)
  - Value stored in computer memory
- **Step 2:** Store it (bind it) to the **left hand side** (the VARIABLE)
  - Retrieve value associated with name by invoking the name (typing it out)

# YOU TRY IT!

- Which of these are allowed in Python? Type them in the console to check.
  - `x = 6`
  - `6 = x`
  - `x*y = 3+4`
  - `xy = 3+4`

# ABSTRACTING EXPRESSIONS

- Why **give names** to values of expressions?
  - To **reuse names** instead of values
  - Makes code easier to read and modify
- Choose variable names wisely
  - Code needs to read
  - Today, tomorrow, next year
  - By you and others
  - You'll be fine if you stick to letters, underscores, don't start with a number

```
#Compute approximate value for pi
```

```
pi = 355/113
```

```
radius = 2.2
```

```
area = pi * (radius**2)
```

```
circumference = pi * (radius*2)
```

comments start with a # and  
are not part of code executed  
– used to tell others what your  
code is doing

an assignment  
\* expression on right  
\* variable name on left

# WHAT IS BEST CODE STYLE?

```
#do calculations  
a = 355/113 * (2.2**2)  
c = 355/113 * (2.2**2)
```

meh

```
p = 355/113  
r = 2.2  
#multiply p with r squared  
a = p*(r**2)  
#multiply p with r times 2  
c = p*(r*2)
```

ok

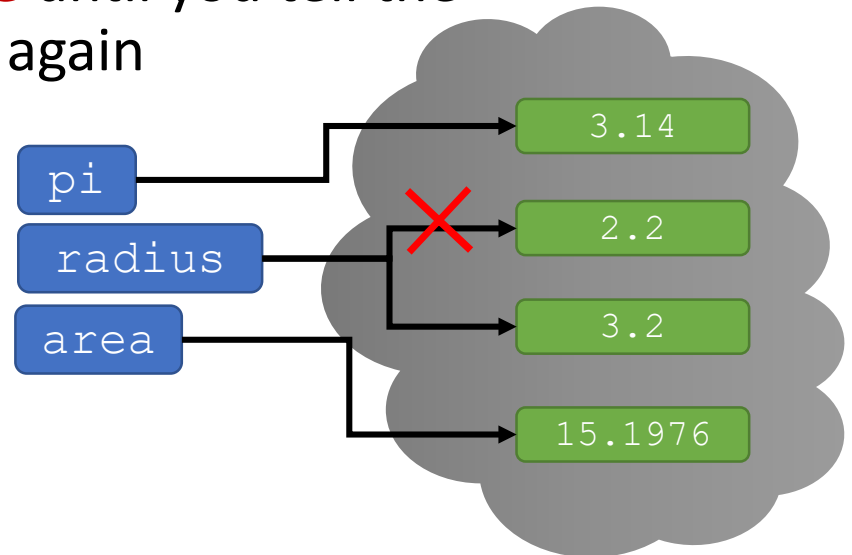
```
#calculate area and circumference of a circle  
#using an approximation for pi  
pi = 355/113  
radius = 2.2  
area = pi*(radius**2)  
circumference = pi*(radius*2)
```

best

# CHANGE BINDINGS

- Can **re-bind** variable names using new assignment statements
- Previous value may still stored in memory but lost the handle for it
- Value for **area does not change** until you tell the computer to do the calculation again

```
pi = 3.14
radius = 2.2
area = pi*(radius**2)
radius = radius+1
```



# BIG IDEA

Lines are evaluated one  
after the other

No skipping around, yet.

We'll see how lines can be skipped/repeated later.

# YOU TRY IT!

- These 3 lines are executed in order. What are the values of `meters` and `feet` variables at each line in the code?

```
meters = 100
```

```
feet = 3.2808 * meters
```

```
meters = 200
```

## ANSWER:

Let's use PythonTutor to figure out what is going on

- [Follow along with this Python Tutor LINK](#)

Where did we tell Python to (re)calculate feet?

# YOU TRY IT!

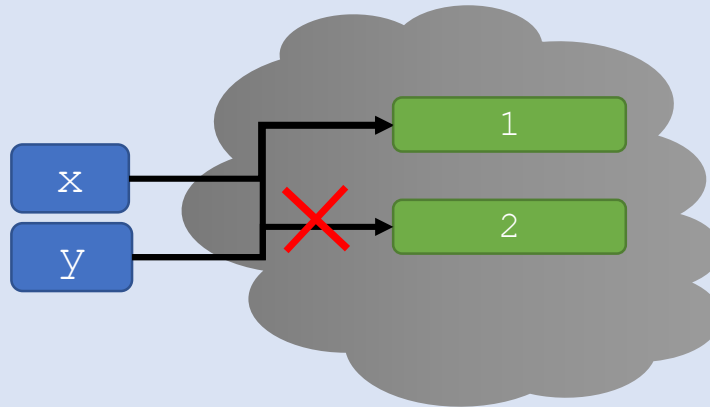
- Swap values of x and y without binding the numbers directly. Debug (aka fix) this code.

```
x = 1
```

```
y = 2
```

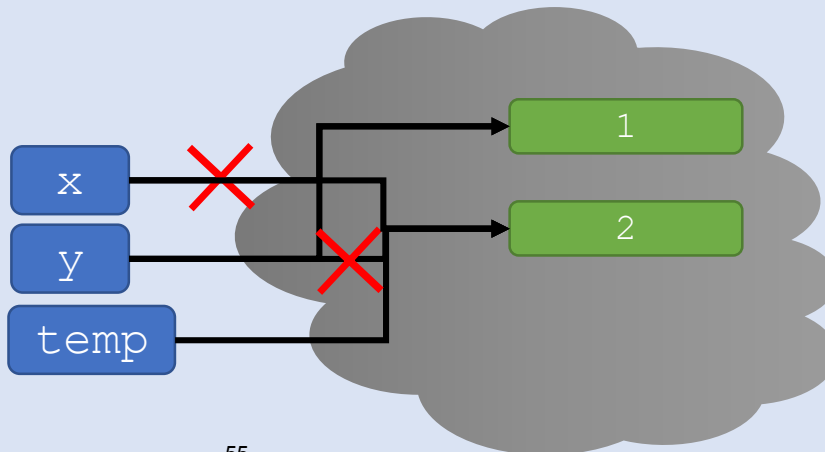
```
y = x
```

```
x = y
```



- [Python Tutor](#) to the rescue?

**ANSWER:**



# SUMMARY

## ■ Objects

- Objects in memory have **types**.
- Types tell Python what **operations** you can do with the objects.
- **Expressions evaluate to one value** and involve objects and operations.
- Variables bind names to objects.
- = sign is an assignment, for ex. `var = type(5*4)`

## ■ Programs

- Programs only **do what you tell them to do**.
- Lines of code are executed **in order**.
- Good variable names and comments help you **read code later**.

MITOpenCourseWare  
<https://ocw.mit.edu>

## 6.100L Introduction to Computer Science and Programming Using Python Fall 2022

For information about citing these materials or our Terms of Use, visit: <https://ocw.mit.edu/terms>.