SI100B: Introduction to Information Science and Technology

Lecture 1

By Ana Bell of MIT

TOPICS

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

- No required textbook
- Read tutorial & reference at https://docs.python.org/
- More tutorials for reference (not required):
  - Chinese: https://www.runoob.com/python3/python3-tutorial.html
  - Basics: https://github.com/jackfrued/Python-100-Days
  - Book: Eric Matthes, Python Crash Course, 3rd ed., 2023

WHAT IS COMPUTATION?

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

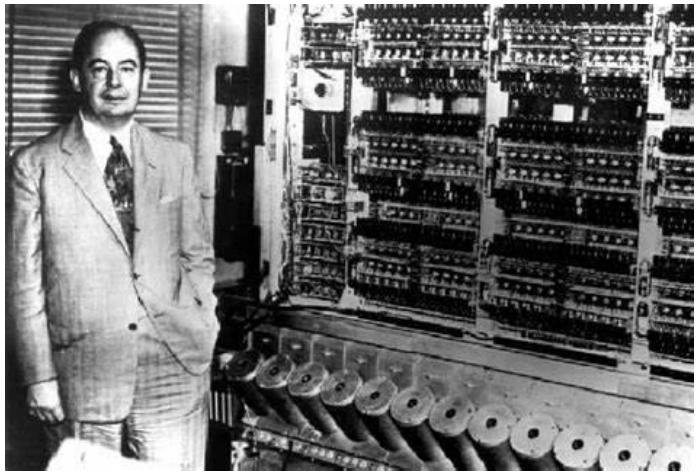
    100s of gigabytes of storage!
- What kinds of calculations?
  - **Built-in** to the machine, e.g., +
  - Ones that **you define** as the programmer

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

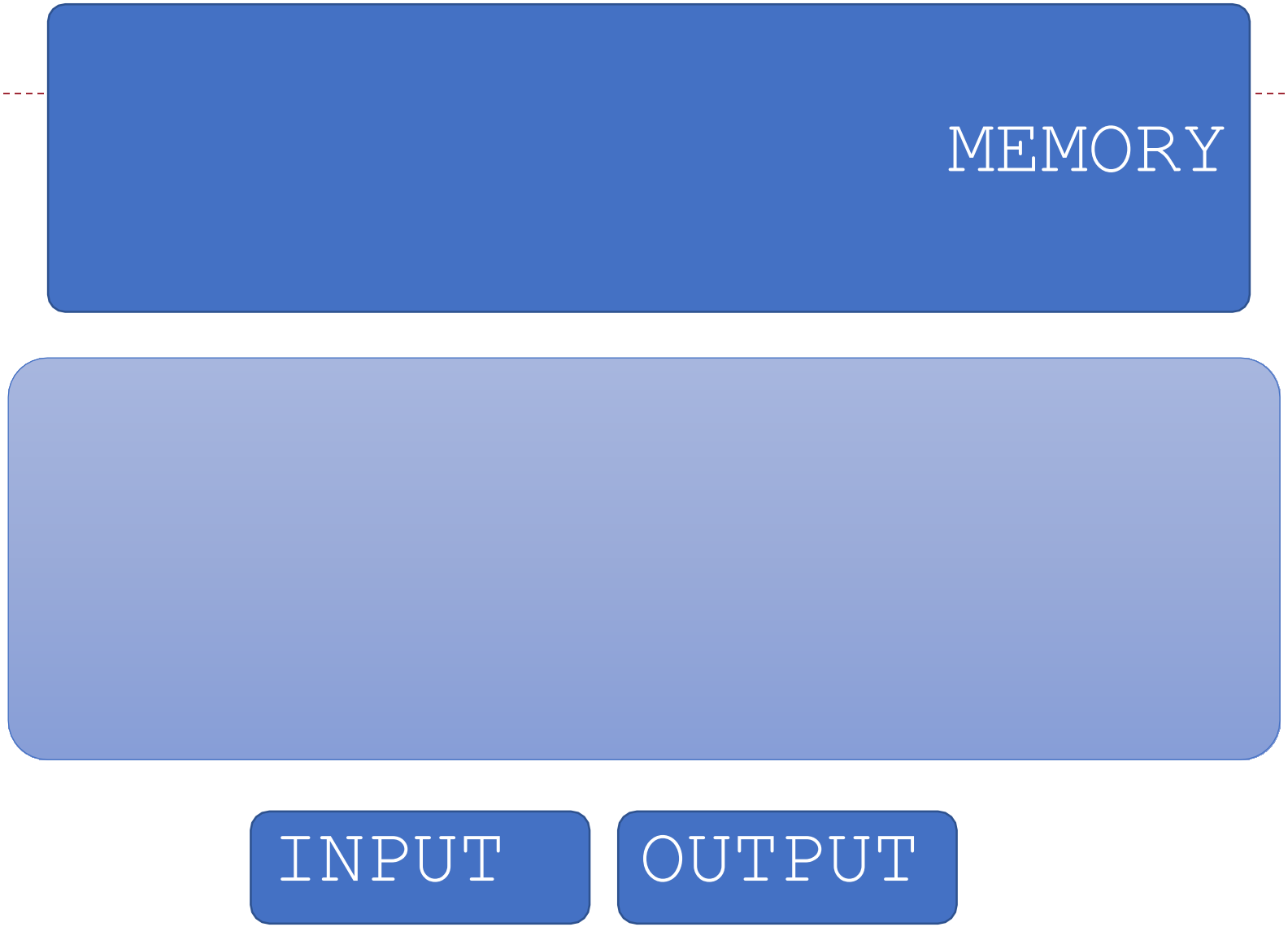
Von Neumann Architecture

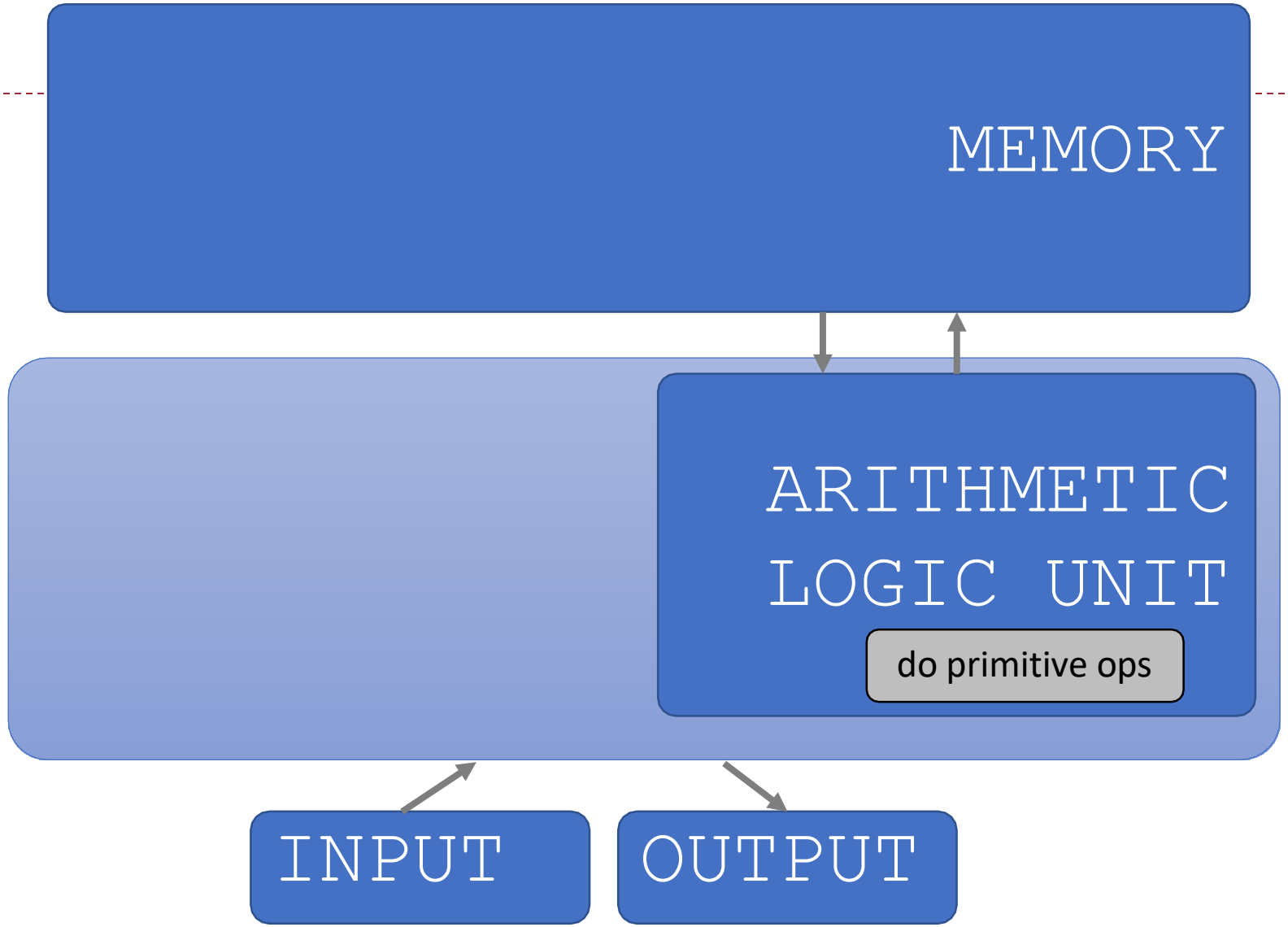
- Proposed by John von Neumann
  - born in Hungry, 1903
  - diploma in chemical engineering at ETH Zurich, 1926
  - PhD in mathematics, Budapest University, 1926
  - postdoc with David Hilbert, 1926-1927
  - affiliated with Princeton University since 1930
- Built upon stored-program computer concept
  - still used nowadays

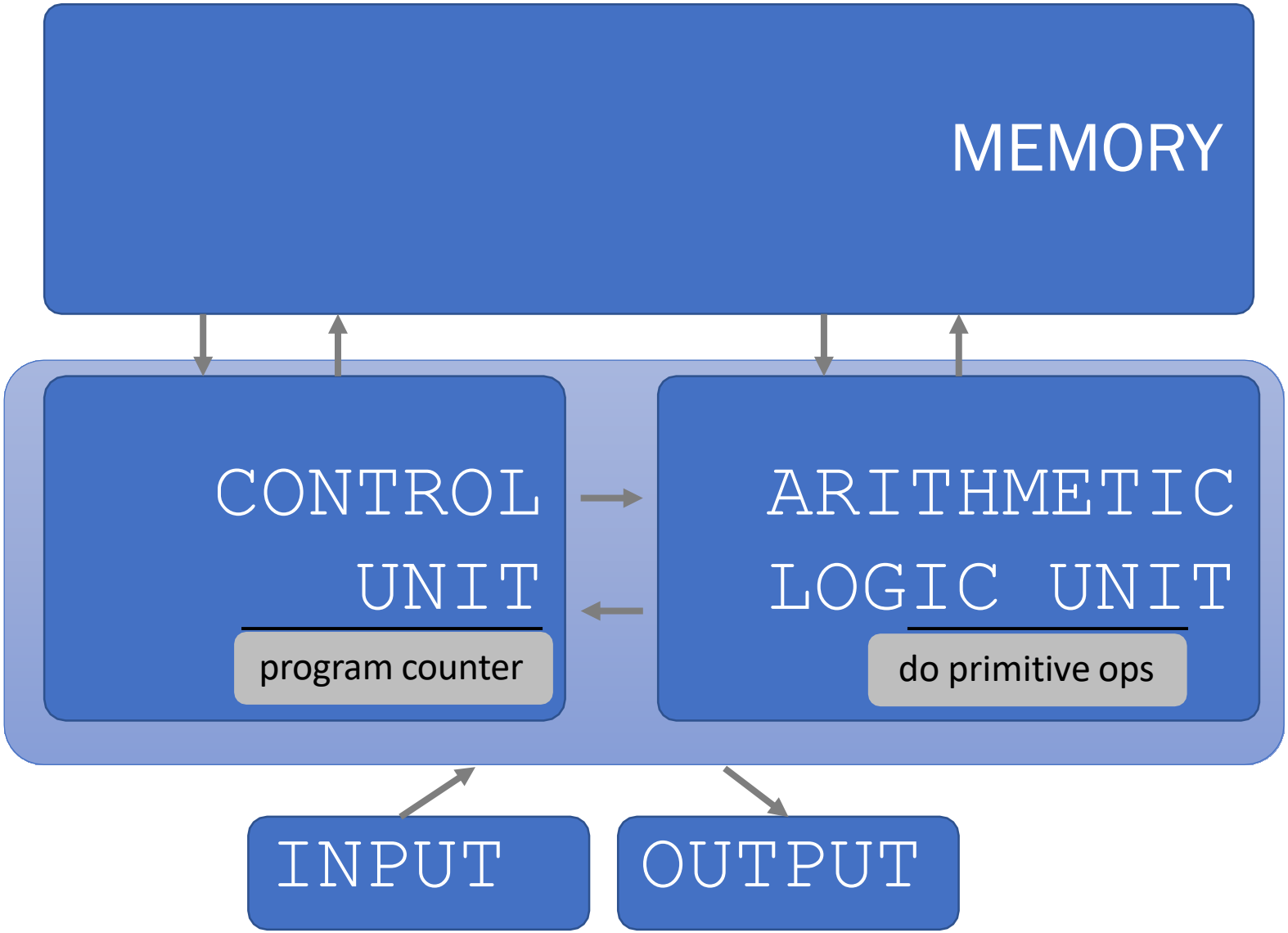


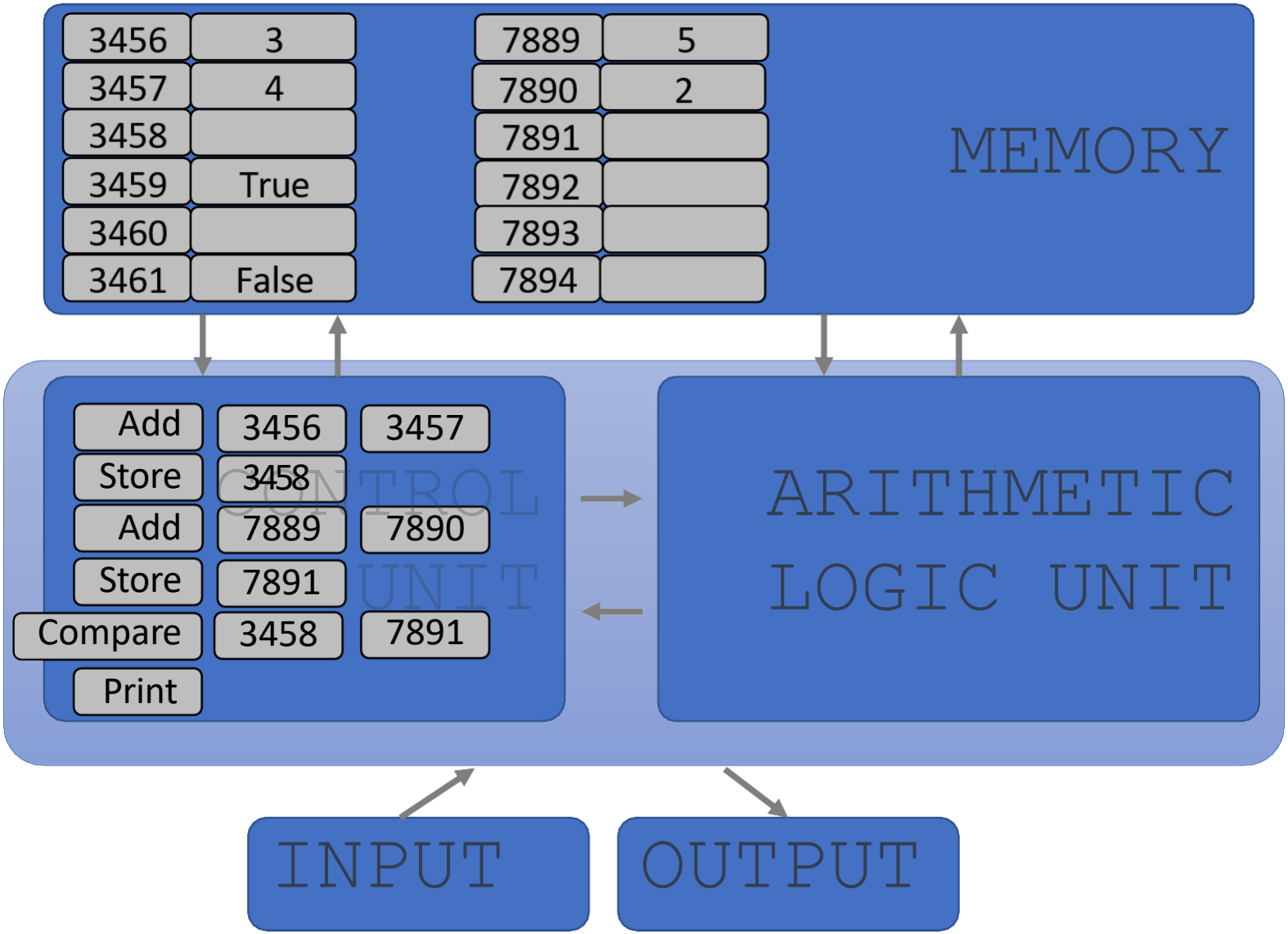
STORED PROGRAM COMPUTER

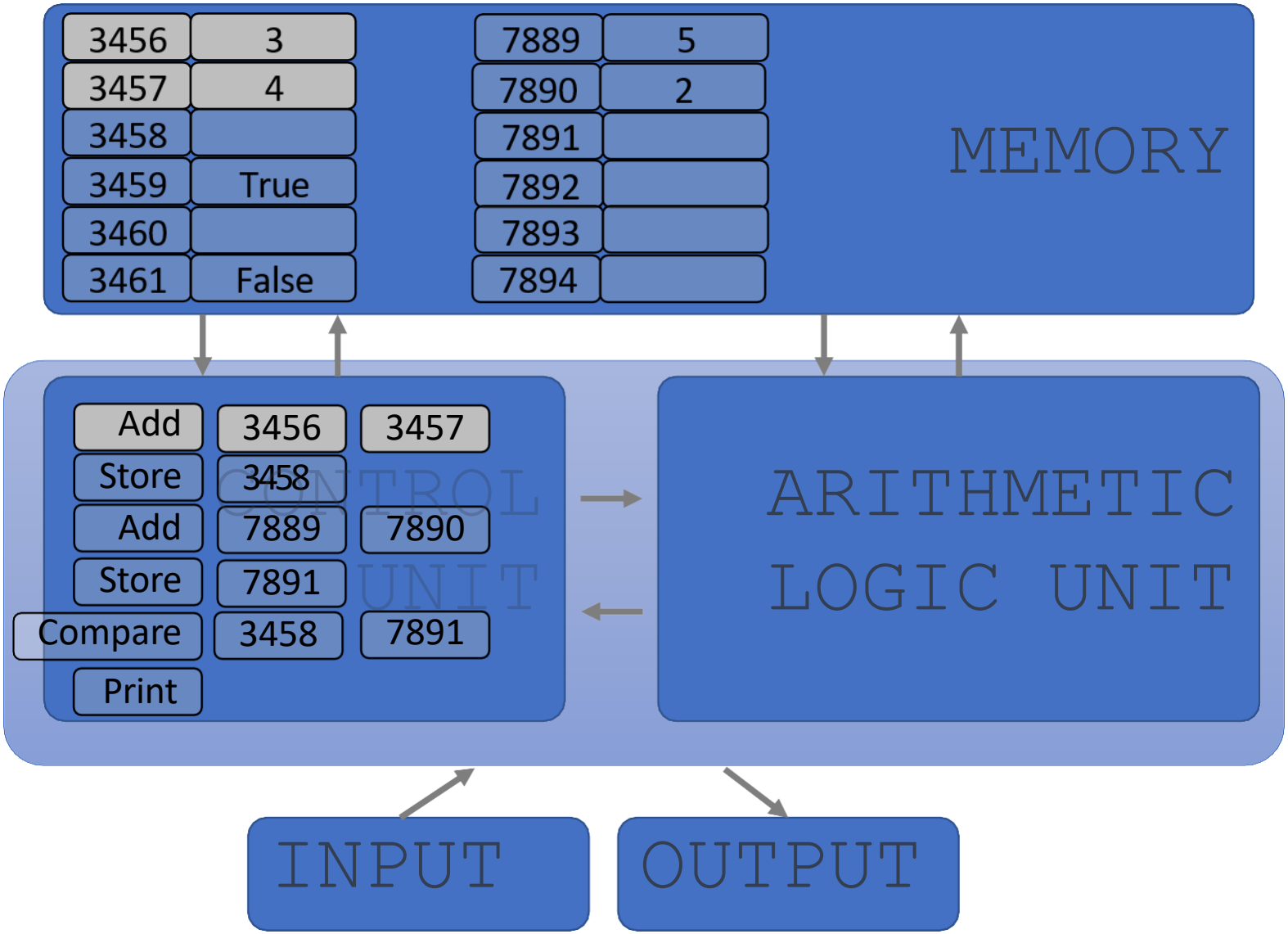
- Sequence of **instructions stored** inside computer
  - Built from predefined set of primitive instructions
  1. Arithmetic and logical
  2. Simple tests
  3. Moving data
- Special program (interpreter) **executes each instruction in order**
  - Use tests to change flow of control through sequence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

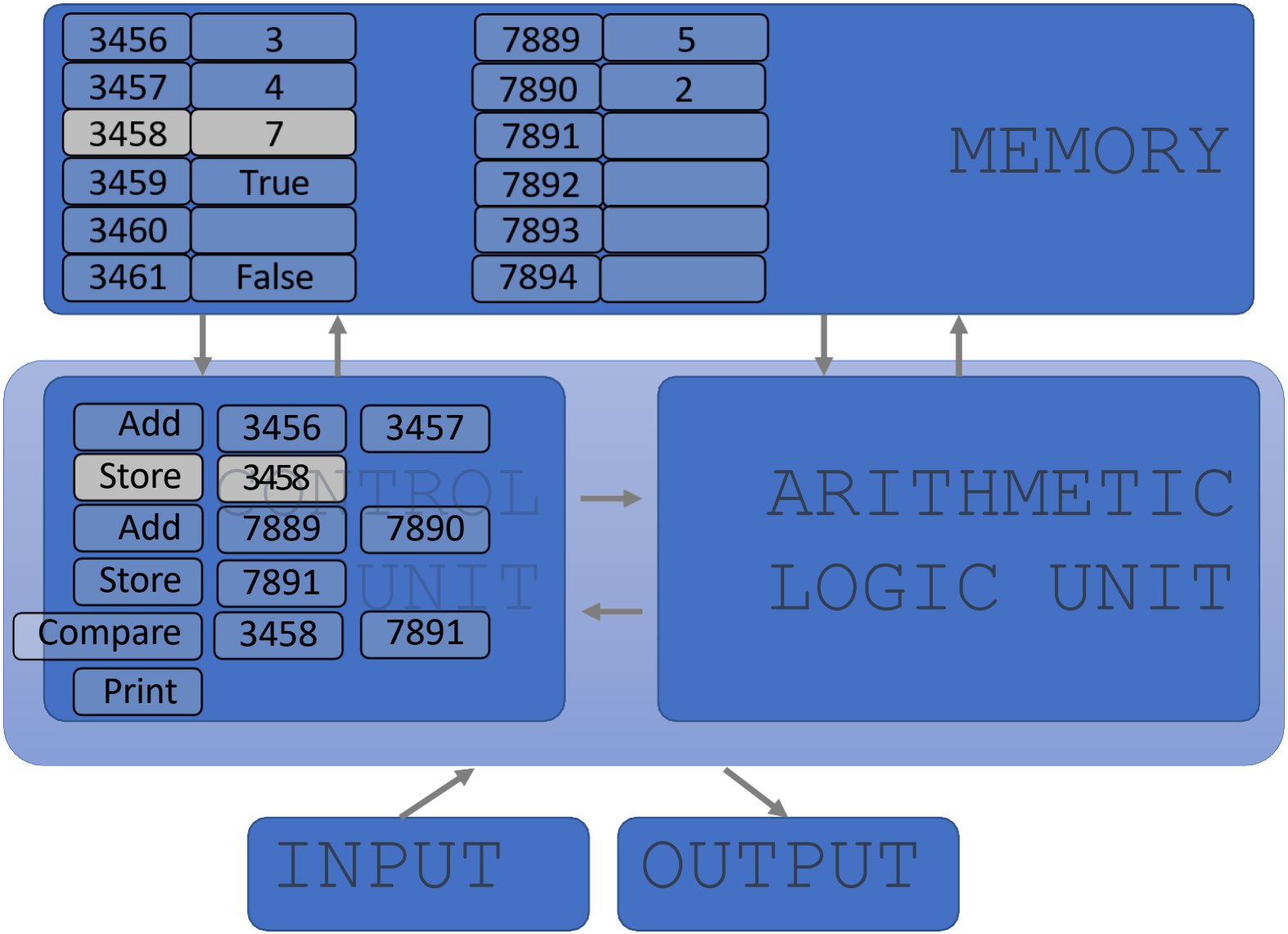

MEMORY
INPUT
OUTPUT

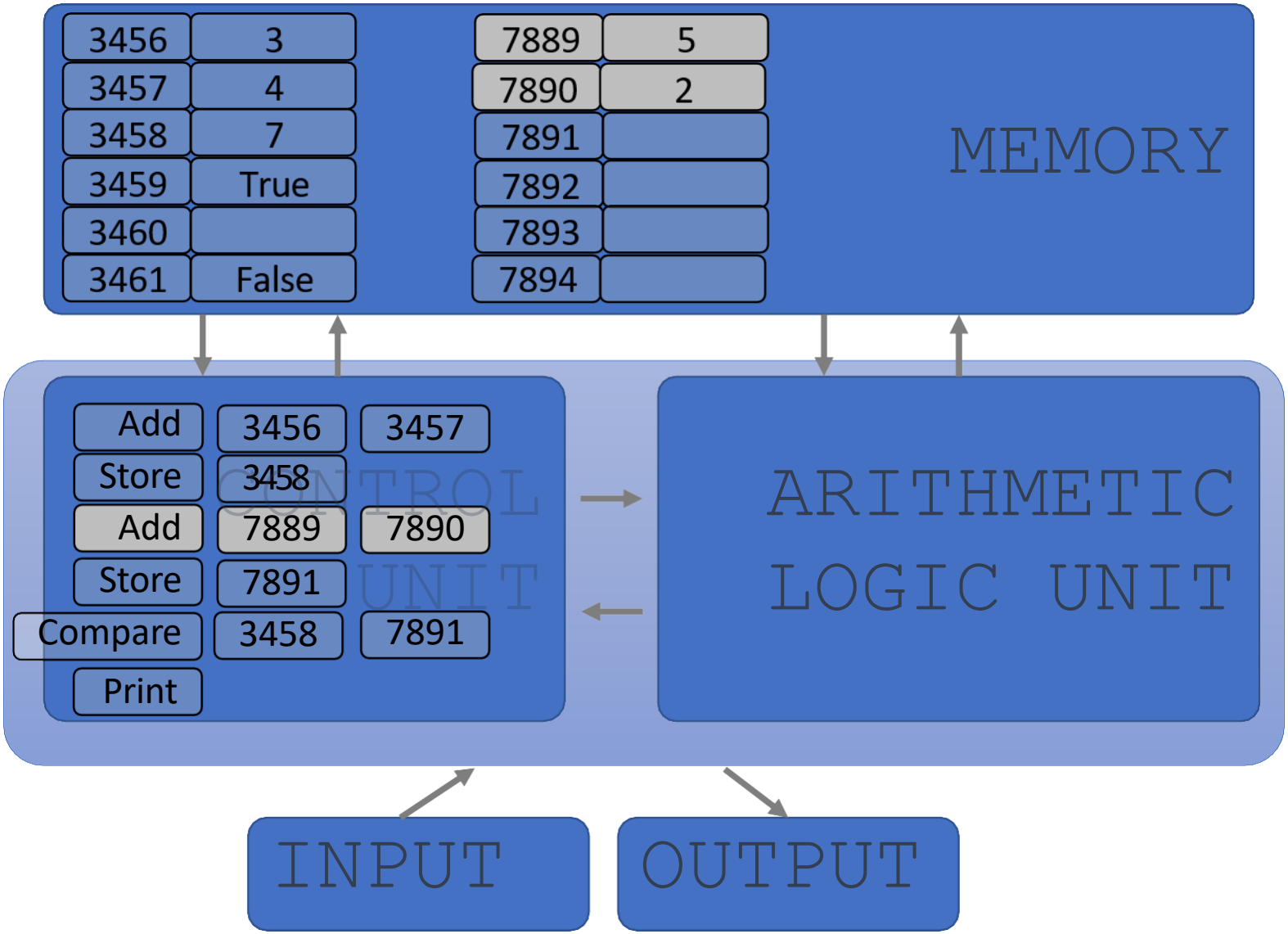

MEMORY
ARITHMETIC
LOGIC UNIT
do primitive ops
INPUT
OUTPUT

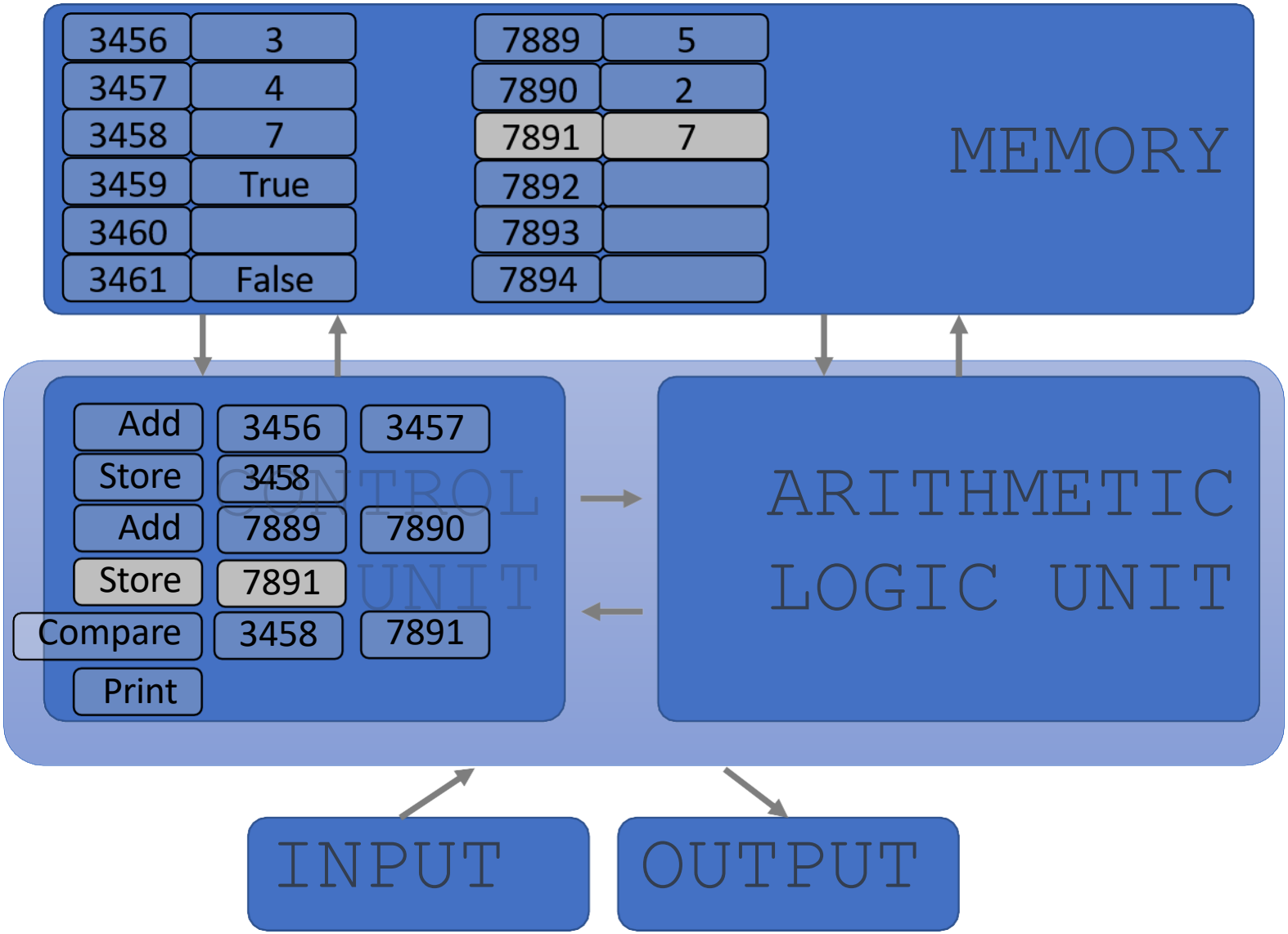

MEMORY
CONTROL UNIT
program counter
ARITHMETIC LOGIC UNIT
do primitive ops
INPUT
OUTPUT

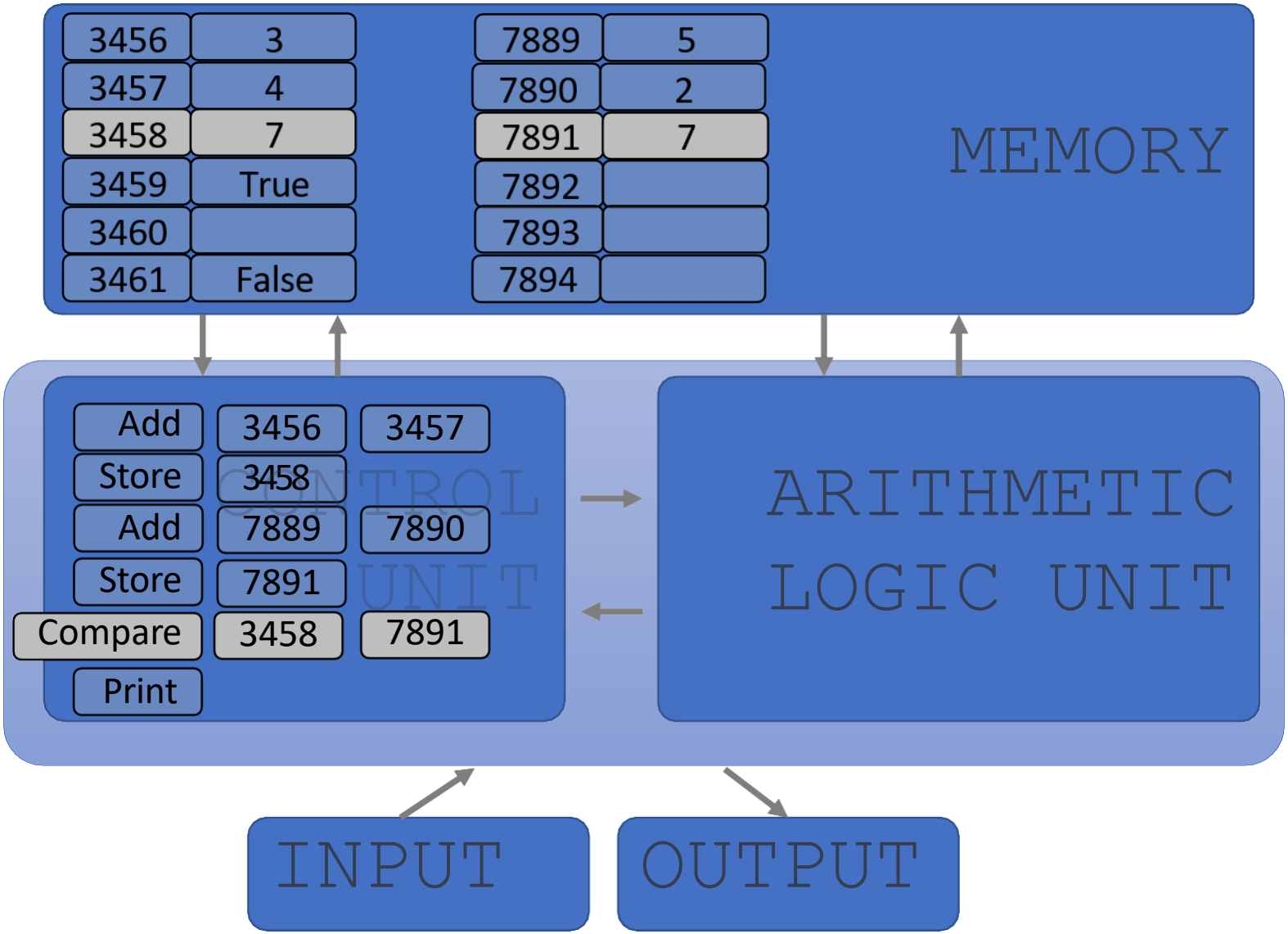

3456
3
3457
4
3458
3459
True
3460
3461
False
7889
5
7890
2
7891
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
CONTROL UNIT
ARITHMETIC LOGIC UNIT
INPUT
OUTPUT

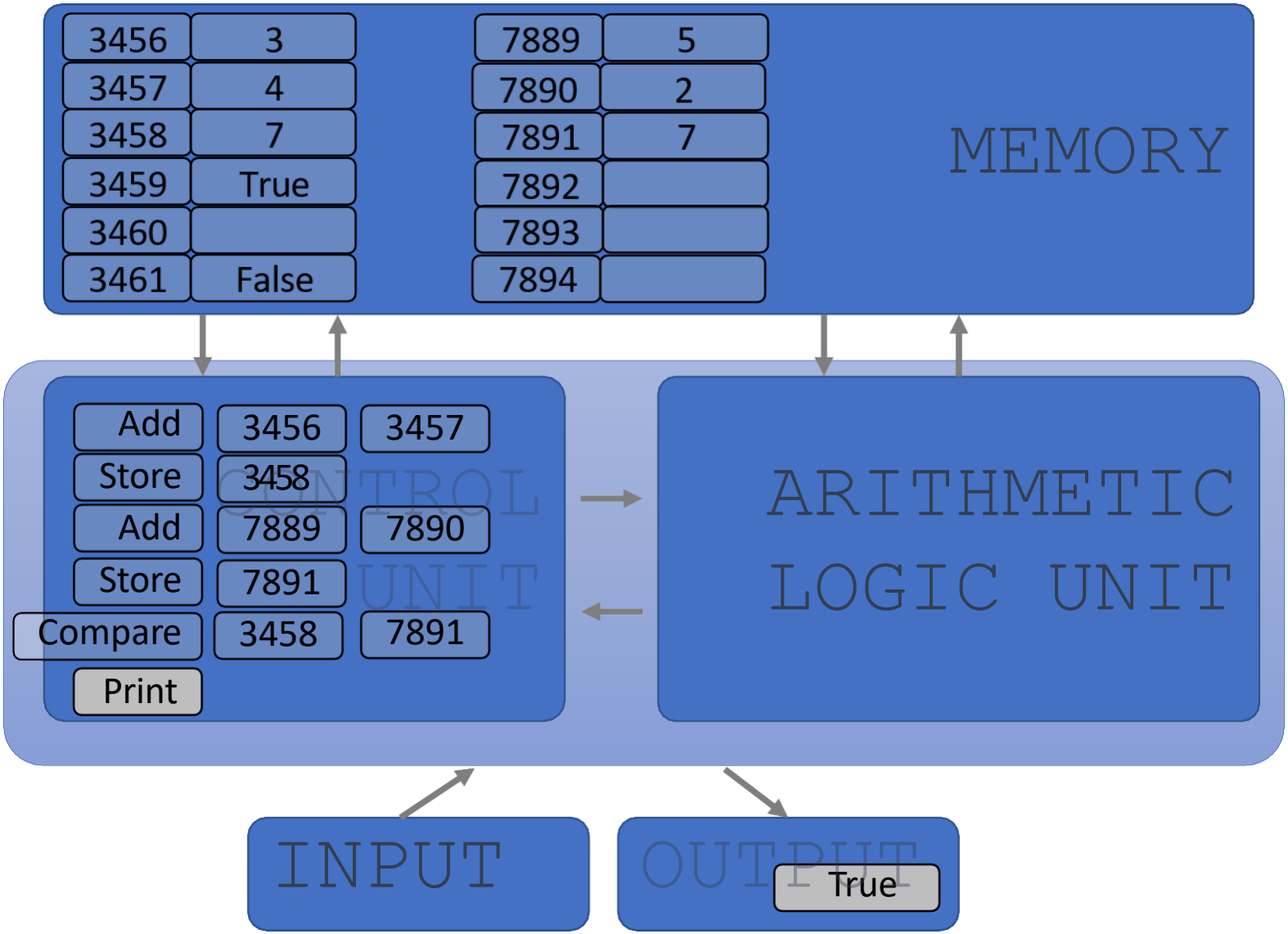

3456
3
3457
4
3458
3459
True
3460
3461
False
7889
5
7890
2
7891
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
CONTROL UNIT
ARITHMETIC LOGIC UNIT
INPUT
OUTPUT


3456
3
3457
4
3458
7
3459
True
3460
3461
False
7889
5
7890
2
7891
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
CONTROL UNIT
ARITHMETIC LOGIC UNIT
INPUT
OUTPUT


3456
3
3457
4
3458
7
3459
True
3460
3461
False
7889
5
7890
2
7891
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
CONTROL UNIT
ARITHMETIC LOGIC UNIT
INPUT
OUTPUT

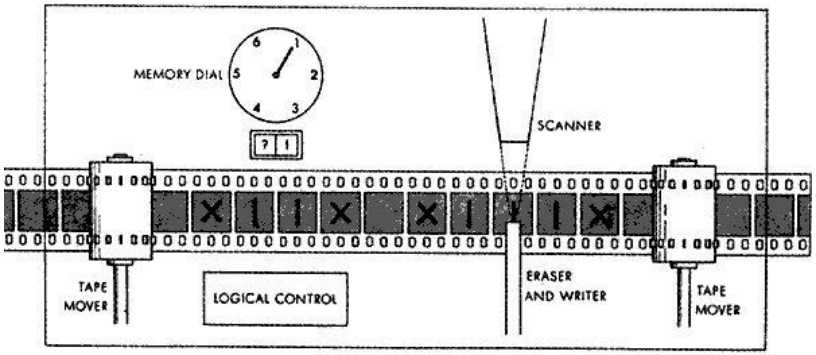

3456
3
3457
4
3458
7
3459
True
3460
3461
False
7889
5
7890
2
7891
7
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
CONTROL UNIT
ARITHMETIC LOGIC UNIT
INPUT
OUTPUT


3456
3
3457
4
3458
7
3459
True
3460
3461
False
7889
5
7890
2
7891
7
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
CONTROL UNIT
ARITHMETIC LOGIC UNIT
INPUT
OUTPUT


3456
3
3457
4
3458
7
3459
True
3460
3461
False
7889
5
7890
2
7891
7
7892
7893
7894
MEMORY
Add
3456
3457
Store
3458
Add
7889
7890
Store
7891
Compare
3458
7891
Print
CONTROL UNIT
ARITHMETIC LOGIC UNIT
INPUT
OUTPUT
True

BASIC PRIMITIVES

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



- 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

PYTHON BASICS

ASPECTS of LANGUAGES

- **Primitive constructs**
  - English: words
  - Chinese: character
  - 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
  - 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**

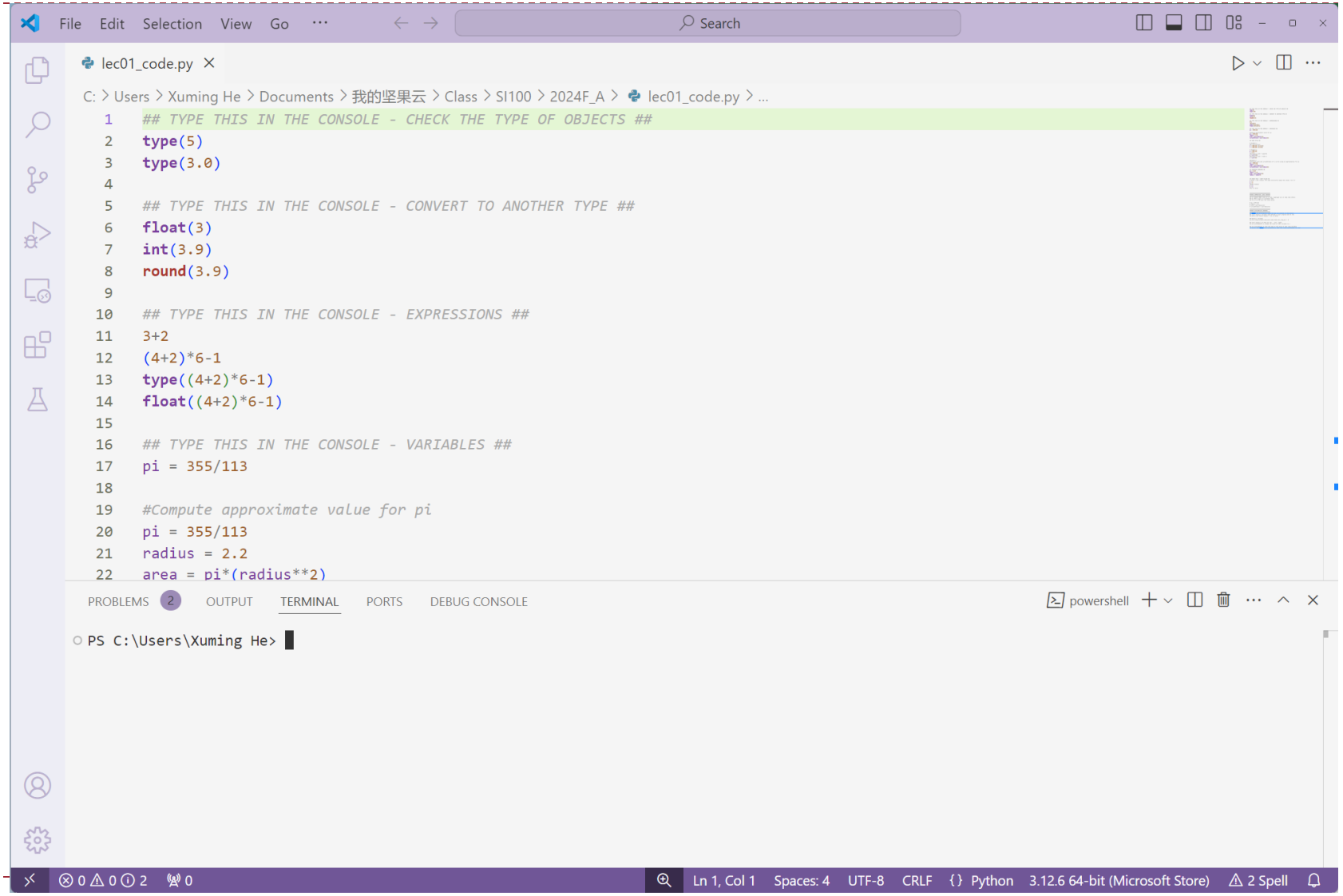
WHERE THINGS GO WRONG

- **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
- Can be typed directly in a **shell** or stored in a **file** that is read into the shell and evaluated

PROGRAMMING ENVIRONMENT: VSCODE



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>`

OPERATORS on int and float

- `i+j` → the **sum**
- `i-j` → the **difference**
- `i*j` → the **product**

if both are ints, result is int
if either or both are floats, result is float

- `i/j` → **division** → result is always a float

- `i//j` → **floor division**
- `i%j` → the **remainder** when `i` is divided by `j`

What is type of output?

- `i**j` → `i` to the **power** of `j`

SIMPLE OPERATIONS

- Parentheses tell Python to do these operations first
  - Like math!
  - `(13-4) / (12*12)`

- **Operator precedence** without parentheses

`**`	
`* / %`	executed left to right, as appear in expression
`+ -`	executed left to right, as appear in expression

VARIABLES

- **Math variables**
  - Abstract
  - Can **represent many values**

```
a + 2 = b - 1
```

```
x * x = y
```

x represents all square roots

- **CS variables** are **different** than math variables
  - Is bound to **one single value** at a given time
  - Can be bound to an expression (but expressions evaluate to one value!)

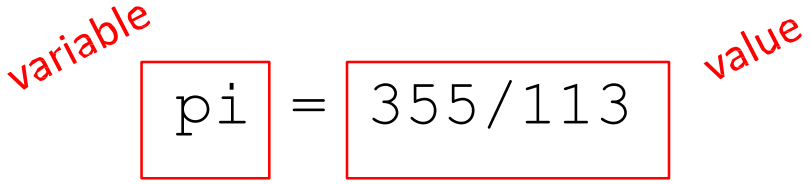
```
m = 10
F = m*9.98
```

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



  - **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

WHAT IS BEST CODE STYLE?

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

meh

# indicates comments. They are not part of the code.

```
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

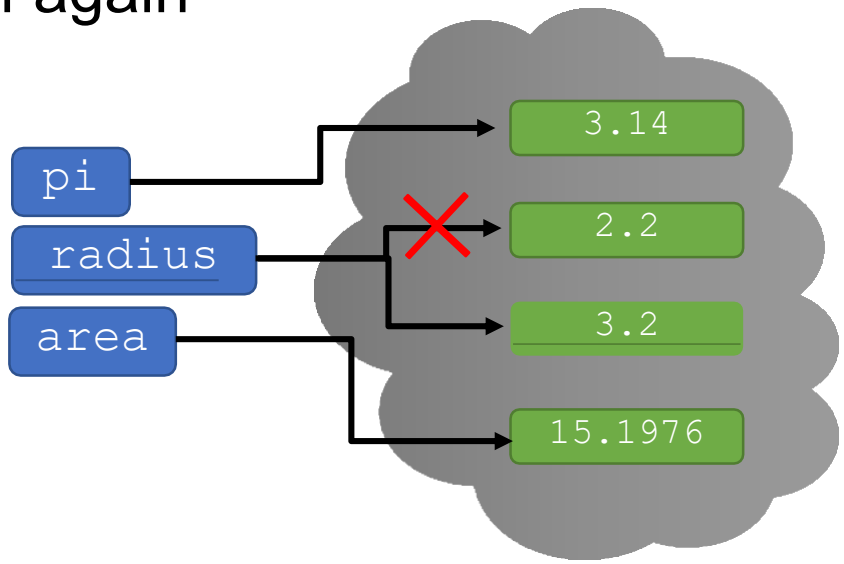
```
#calculate area and circumference of a circle
#using an approximation for pi
pi = 355/113
radius = 2.2
area = pi*(radius**2)
circumference = pi*(radius*2)
```

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.

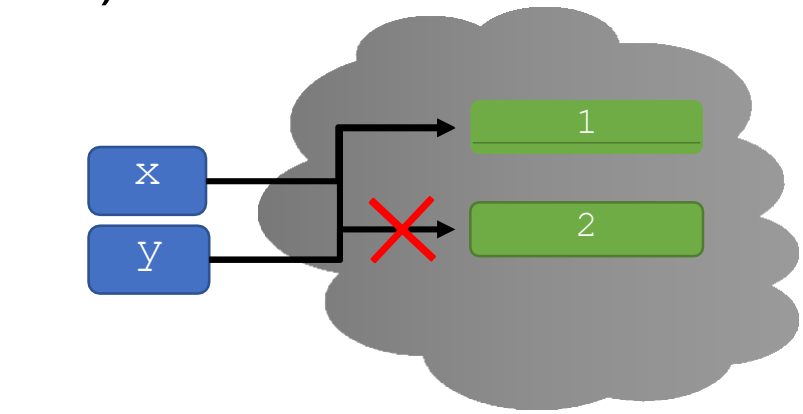
YOU TRY IT!

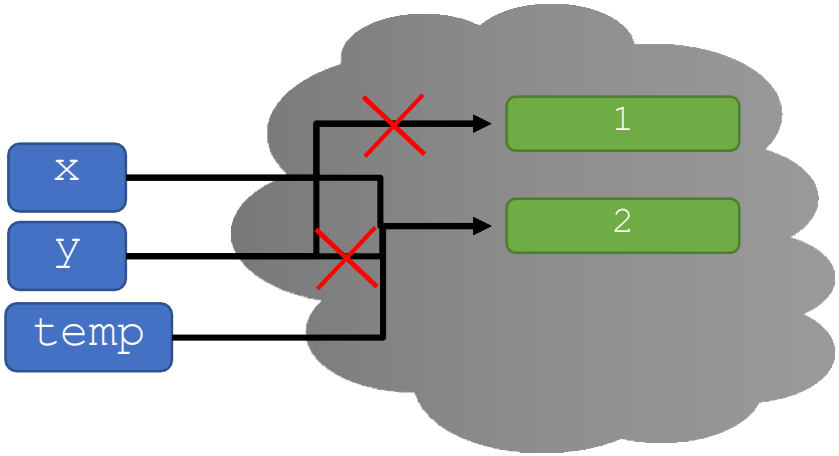
- Swap values of x and y without binding the numbers directly. Debug (aka fix) this code.

```
x = 1
y = 2

y = x
x = y
```

ANSWER:





STRINGS

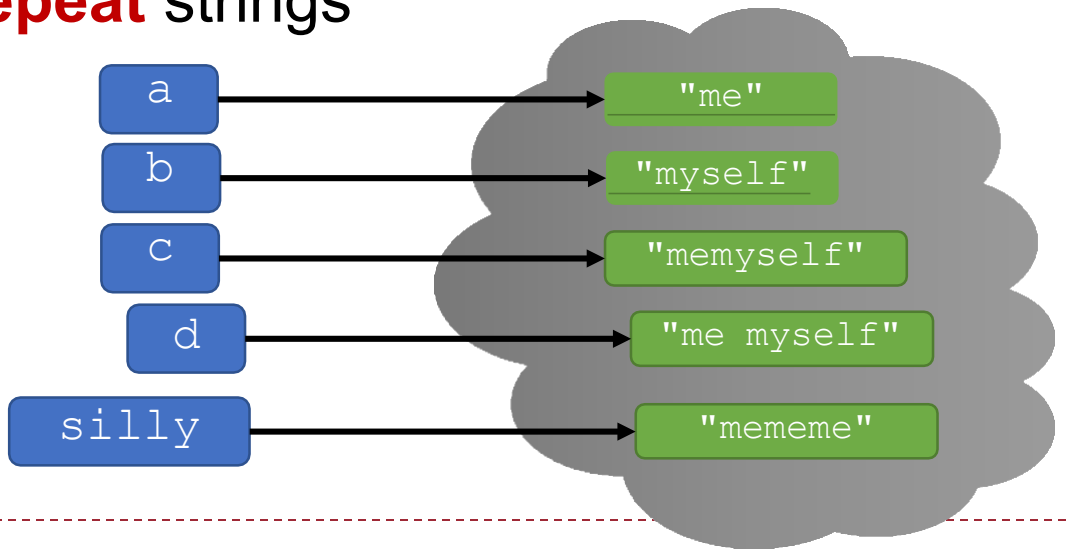
STRINGS

- Think of a str as a **sequence** of case sensitive characters
  - Letters, special characters, spaces, digits
- Enclose in **quotation marks or single quotes**
  - Just be consistent about the quotes

```
a = "me"
z = 'you'
```

- **Concatenate** and **repeat** strings

```
b = "myself"
c = a + b
d = a + " " + b
silly = a * 3
```



YOU TRY IT!

- What's the value of s1 and s2?
- ```
  b = ":"
  c = ")"
  s1 = b + 2*c
  ```
- ```
  f = "a"
  g = " b"
  h = "3"
  s2 = (f+g)*int(h)
  ```

STRING OPERATIONS

- len() is a function used to retrieve the **length** of a string in the parentheses

```
s = "abc"
len(s)    →    evaluates to 3
chars = len(s)
```

Expression that evaluates to 3

SLICING to get ONE CHARACTER IN A STRING

- Square brackets used to perform **indexing** into a string to get the value at a certain index/position

```
s = "abc"
```

index: 0 1 2 ← indexing always starts at 0
index: -3 -2 -1 ← index of last element is len(s) - 1 or -1

`s[0]` → evaluates to "a"
`s[1]` → evaluates to "b"
`s[2]` → evaluates to "c"
`s[3]` → trying to index out of bounds, error
`s[-1]` → evaluates to "c"
`s[-2]` → evaluates to "b"
`s[-3]` → evaluates to "a"

SLICING to get a SUBSTRING

- Can **slice** strings using [start:stop:step]
- Get characters at **start**
- up to and including **stop-1**
- taking every **step** characters

This is confusing as you are starting out :(Can't go wrong with explicitly giving start, stop, end every time.

- If give two numbers, [start:stop], step=1 by default
- If give one number, you are back to indexing for the character at one location (prev slide)
- You can also omit numbers and leave just colons (try this out!)

SLICING EXAMPLES

- Can **slice** strings using [start:stop:step]
- Look at step first. +ve means go left-to-right
  -ve means go right-to-left

```
s = "abcdefgh"
```

index: 0 1 2 3 4 5 6 7
index: -8 -7 -6 -5 -4 -3 -2 -1

If unsure what some command does, try it out in your console!

`s[3:6]` → evaluates to `"def"`, same as `s[3:6:1]`

`s[3:6:2]` → evaluates to `"df"`

`s[:]` → evaluates to `"abcdefgh"`, same as `s[0:len(s):1]`

`s[::-1]` → evaluates to `"hgfedcba"`

`s[4:1:-2]` → evaluates to `"ec"`

YOU TRY IT!

```
s = "ABC d3f ghi"

s[3:len(s)-1]
s[4:0:-1]
s[6:3]
```

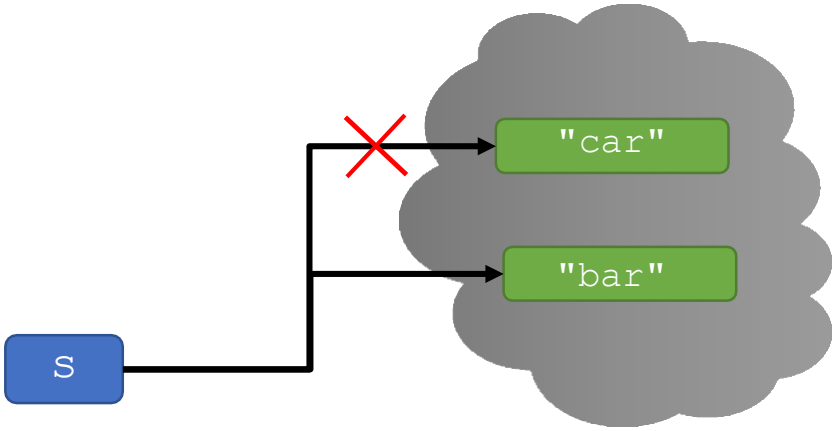
IMMUTABLE STRINGS

- Strings are "**immutable**" – cannot be modified
- You can create **new objects** that are versions of the original one
- Variable name can only be bound to one object

```
s = "car"
s[0] = 'b'
s = 'b'+s[1:len(s)]
```

→ gives an error

→ is allowed,
s bound to new object



BIG IDEA

If you are wondering "what happens if"…

Just try it out in the console!