

Programming Principles 2

Lecture 1

TOPICS

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

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

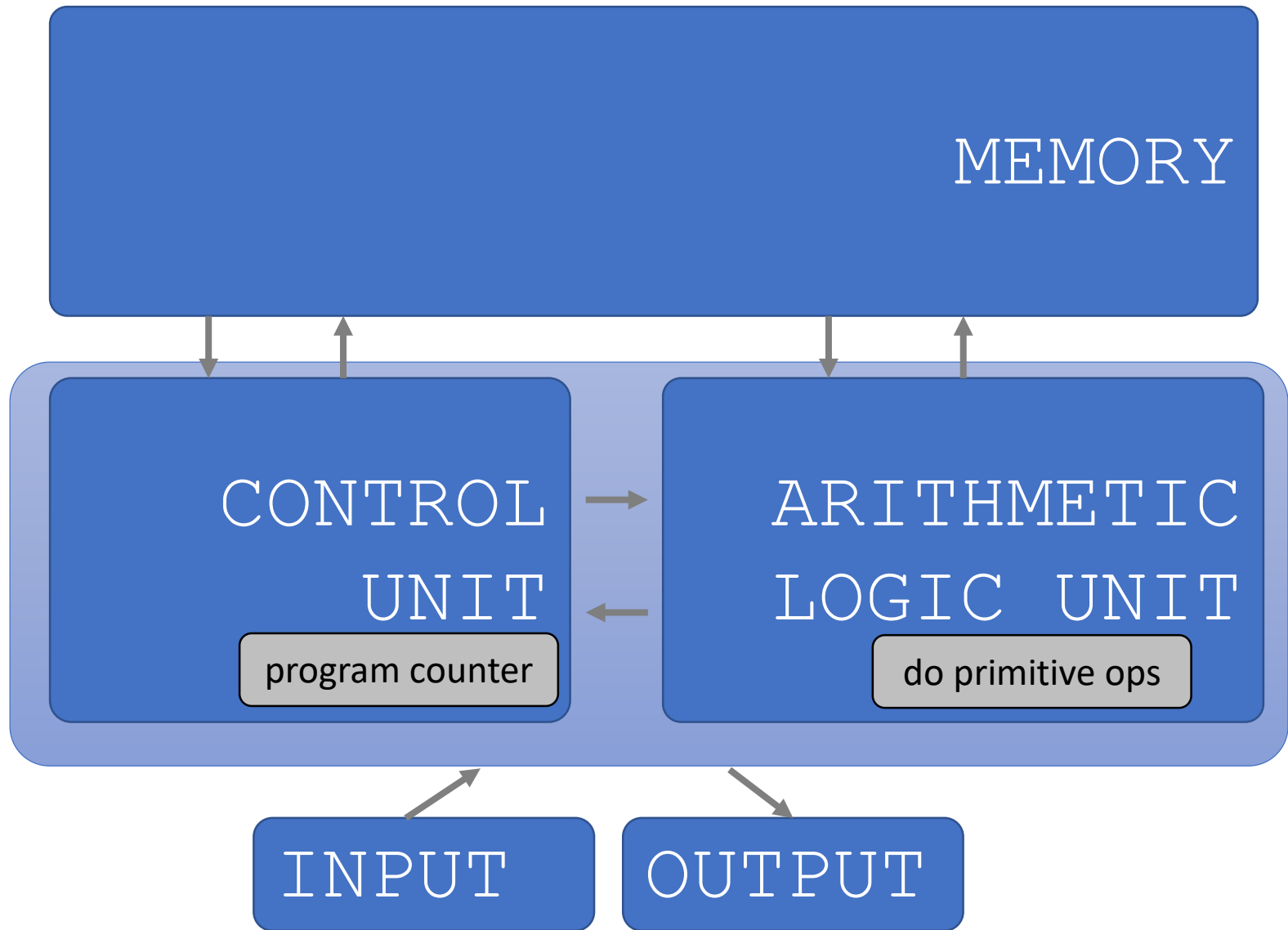
- Two things computers do:
 - Performs simple **operations**
100s of billions per second!
 - **Remembers** results
100s of gigabytes of storage!
- What kinds of calculations?
 - **Built-in** to the machine, e.g., +
 - Ones that **you define** as the programmer
- The BIG IDEA here?

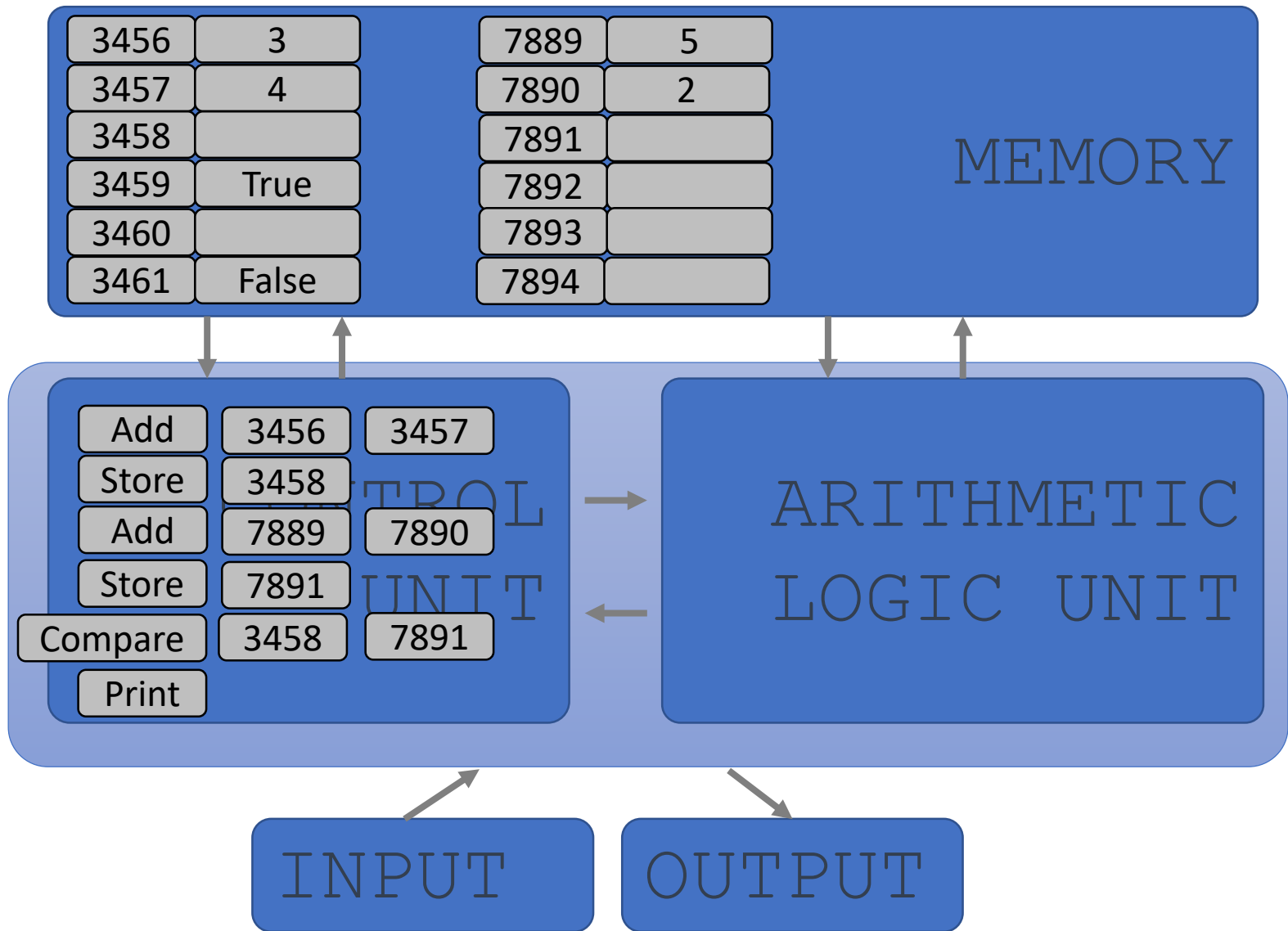
COMPUTERS are MACHINES that EXECUTE ALGORITHMS

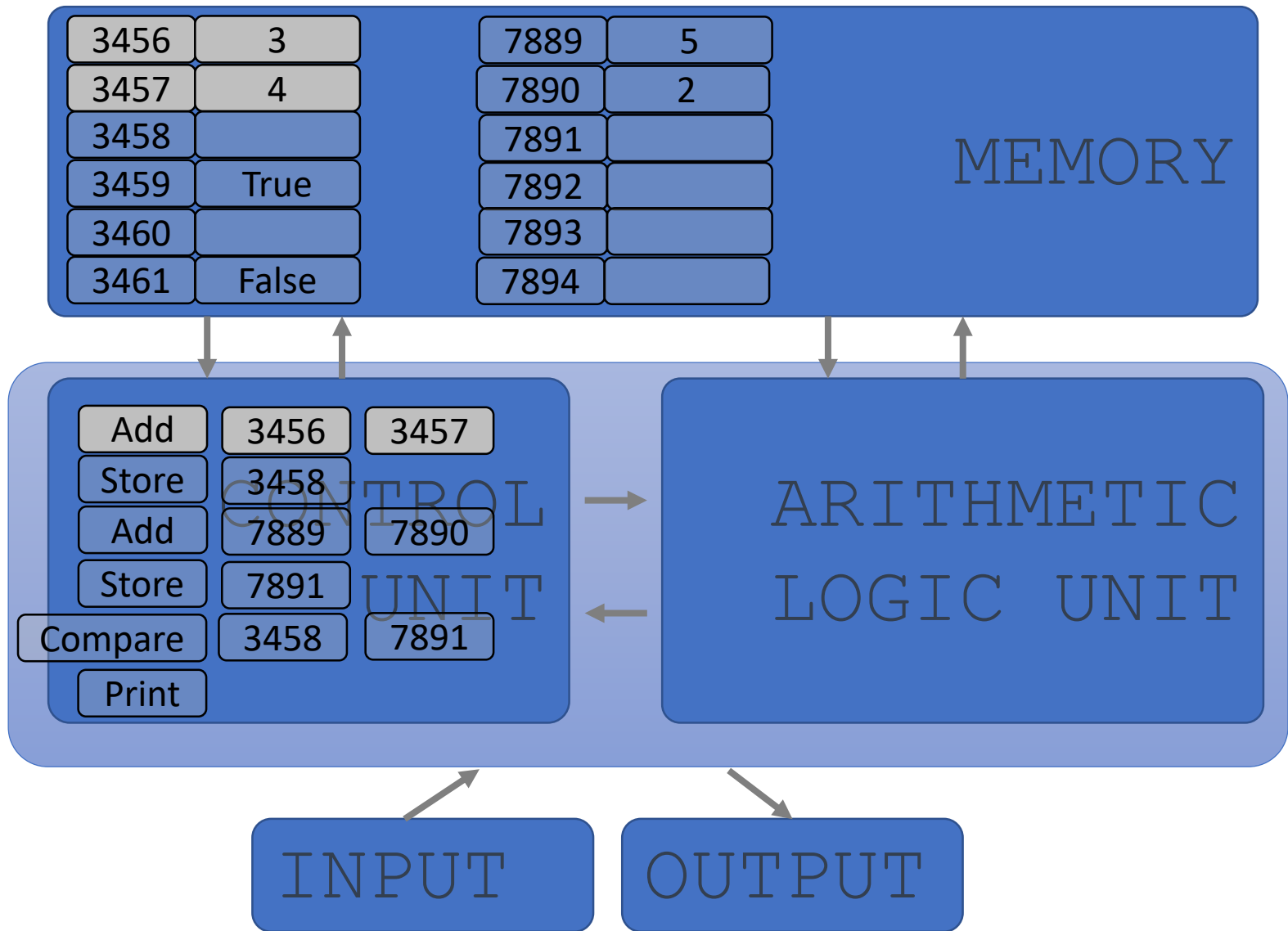
- **Fixed program** computer
 - Fixed set of algorithms
 - What we had until 1940's
- **Stored program** computer
 - Machine stores and executes instructions
- **Key insight:** Programs are no different from other kinds of data

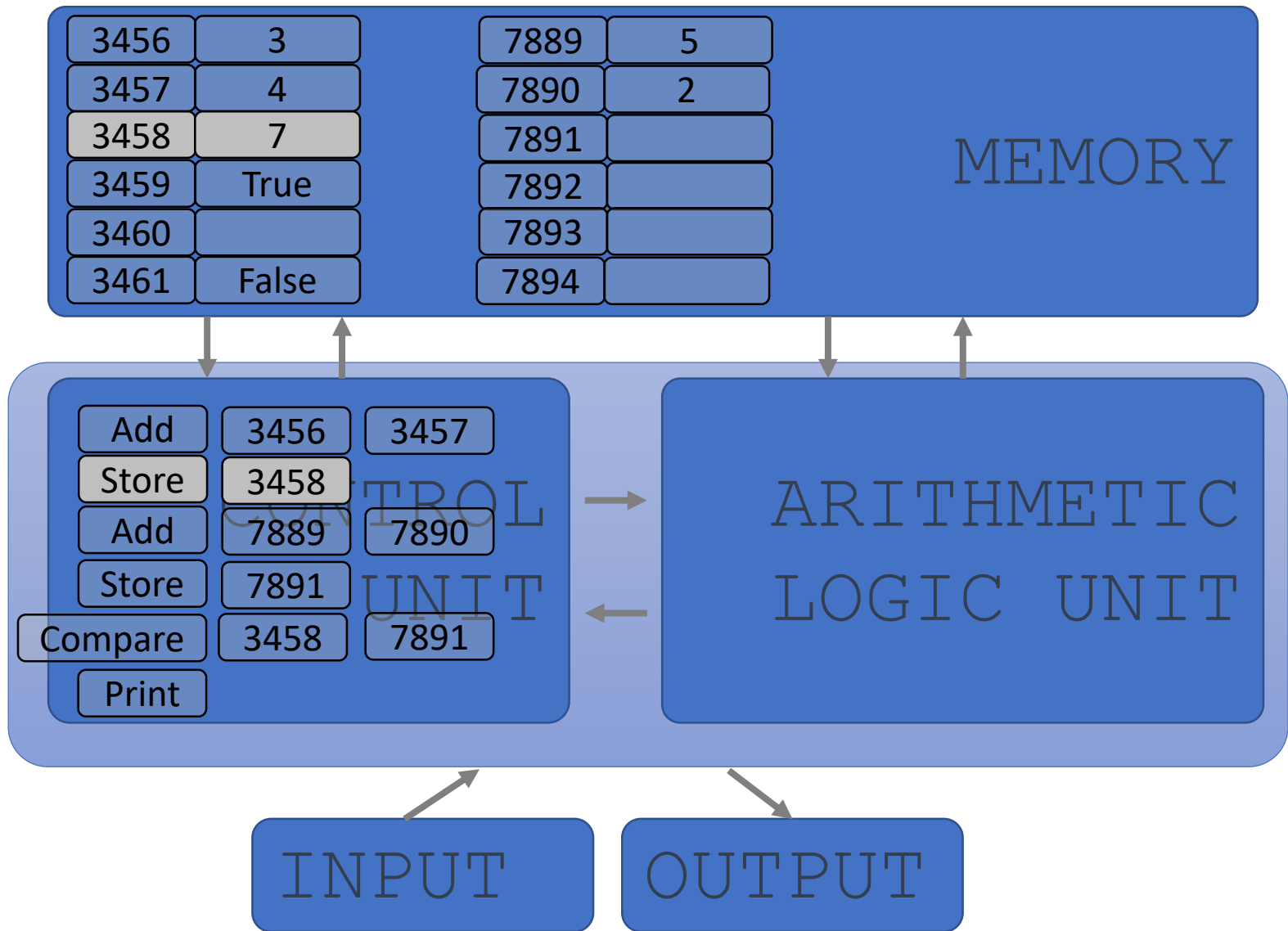
STORED PROGRAM COMPUTER

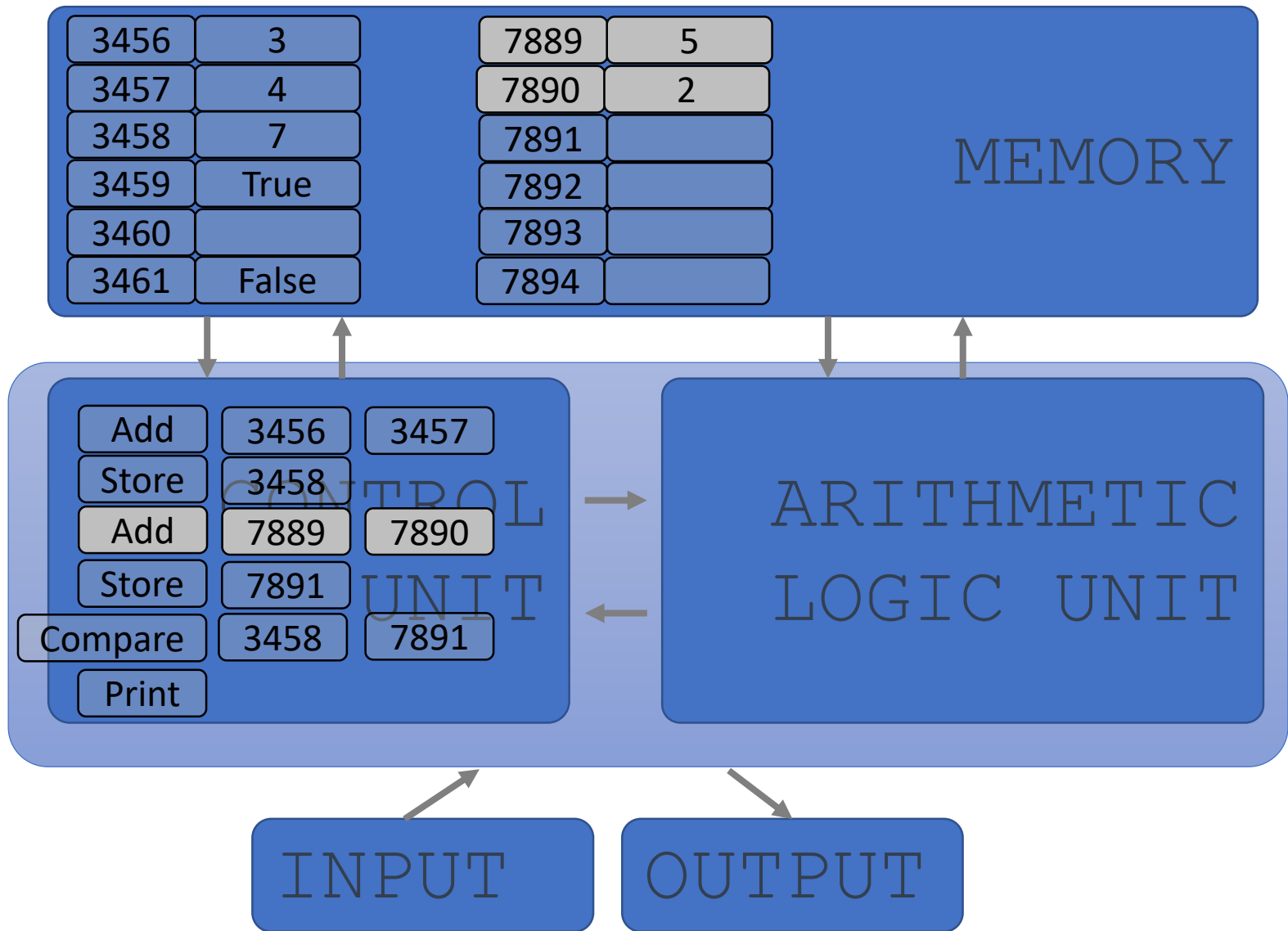
- Sequence of **instructions stored** inside computer
 - Built from predefined set of primitive instructions
 - 1) Arithmetic and logical
 - 2) Simple tests
 - 3) Moving data
- Special program (interpreter) **executes each instruction in order**
 - Use tests to change flow of control through sequence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

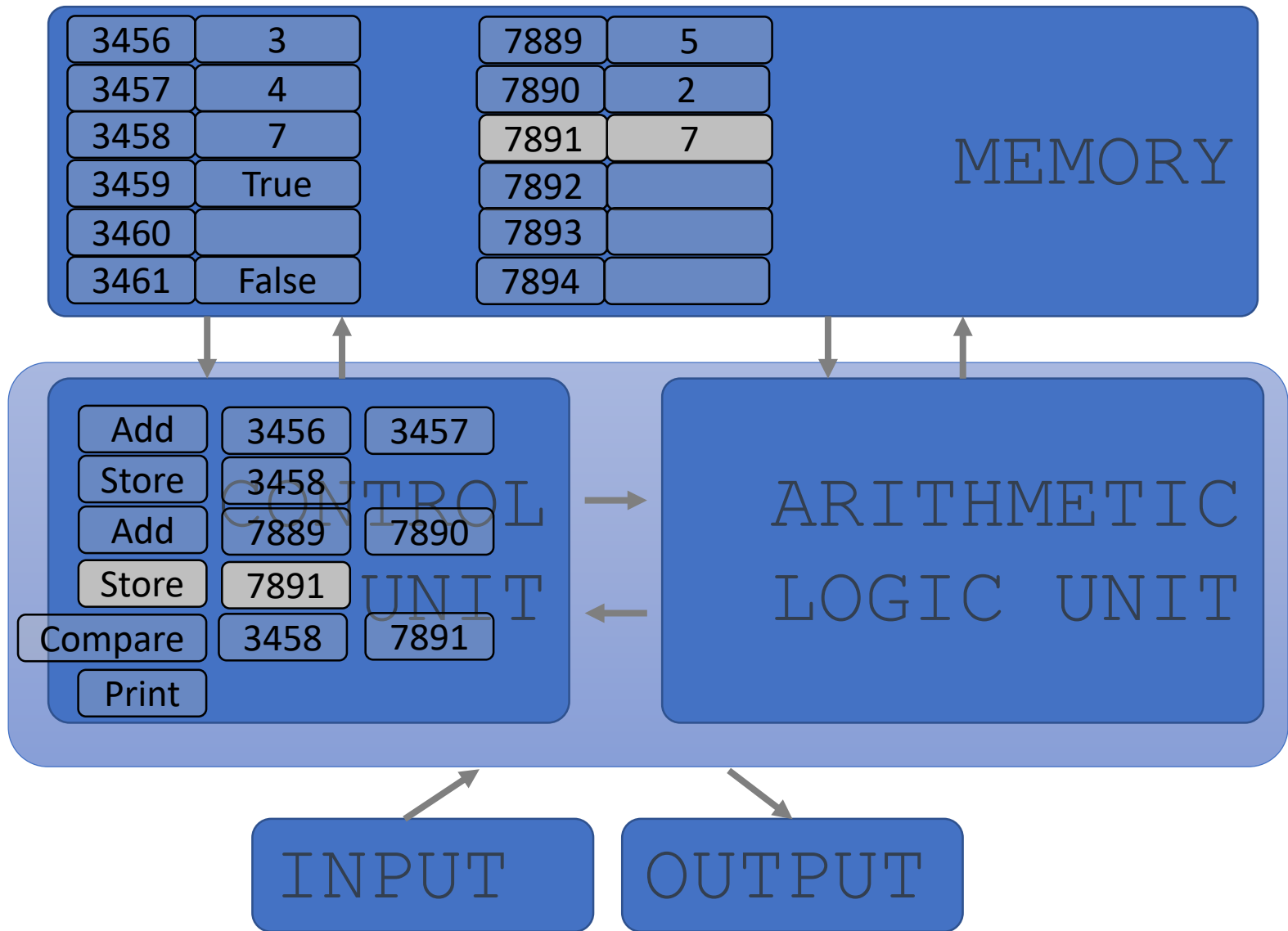


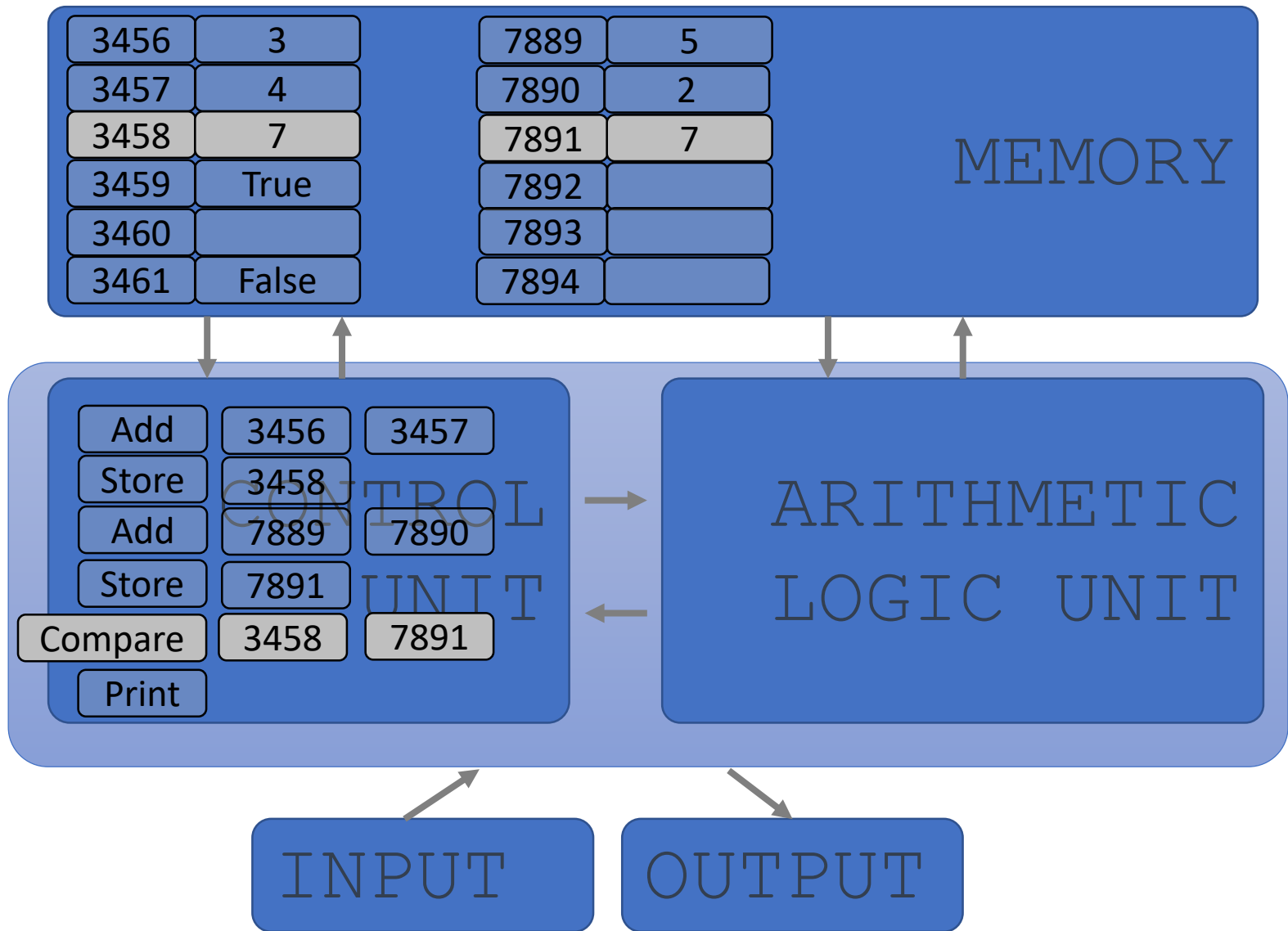


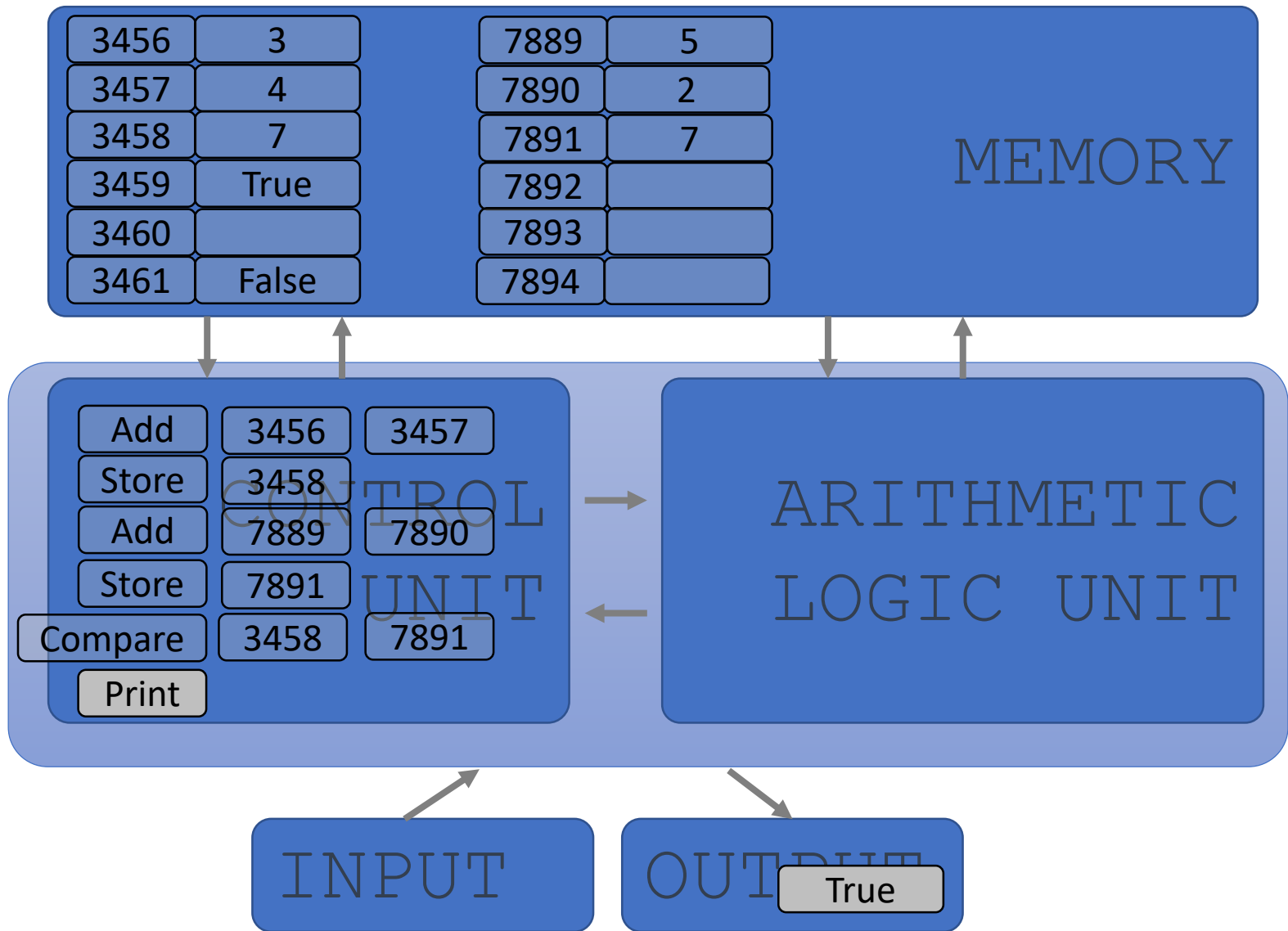






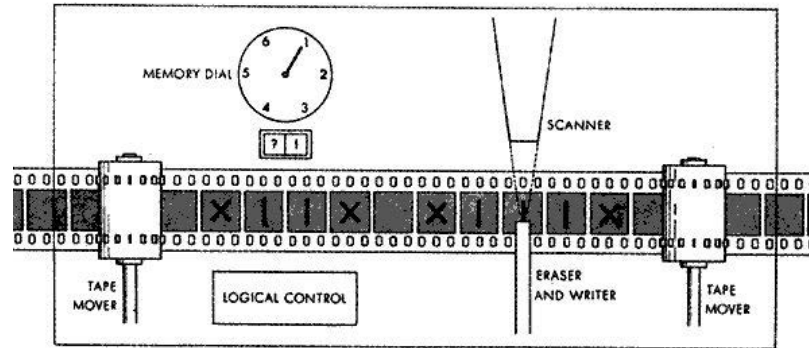






BASIC PRIMITIVES

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



- 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

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
```

*what you write into the
Python shell*

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

```
float
```

*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`

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

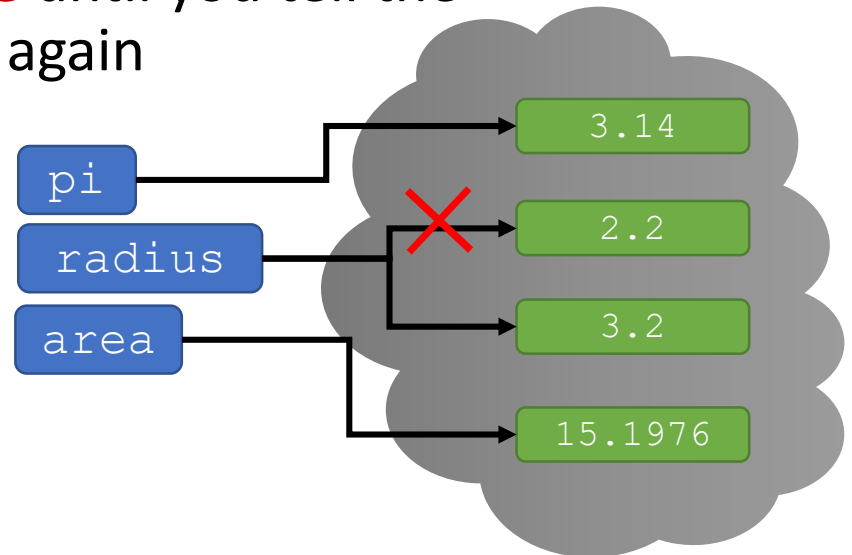
```
#calculate area and circumference of a circle  
#using an approximation for pi  
pi = 355/113  
radius = 2.2  
area = pi*(radius**2)  
circumference = pi*(radius*2)
```

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!

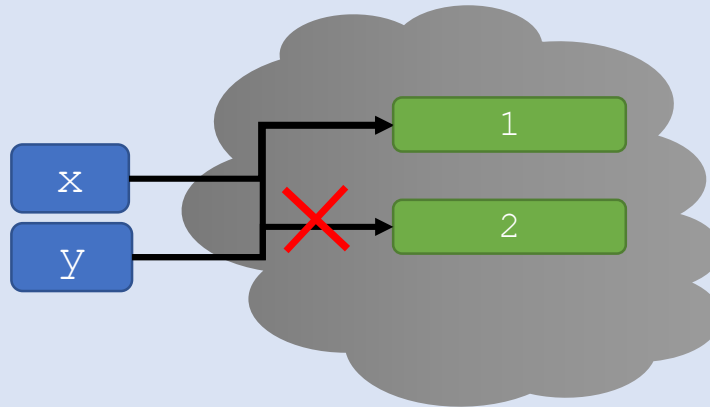
- Swap values of x and y without binding the numbers directly. Debug (aka fix) this code.

```
x = 1
```

```
y = 2
```

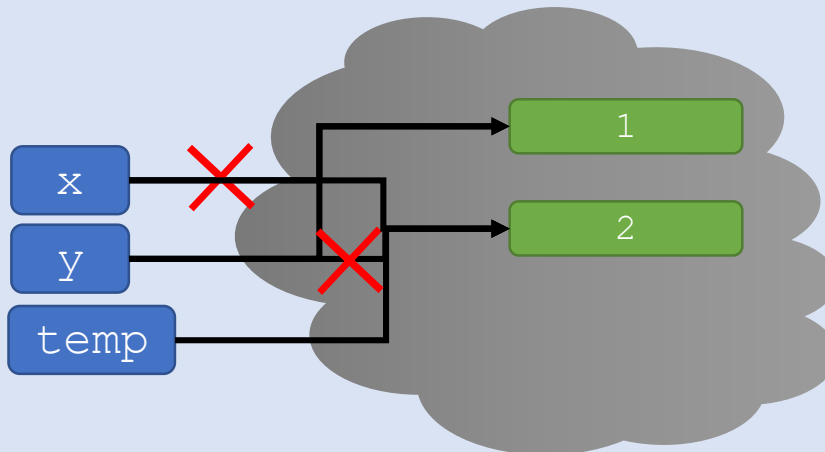
```
y = x
```

```
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**.

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'
```

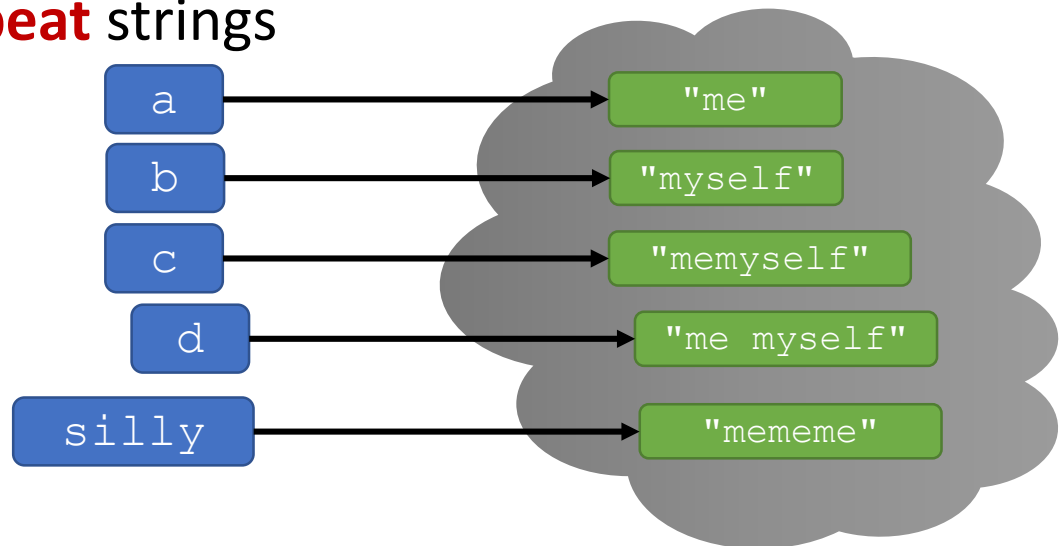
- **Concatenate** and **repeat** strings

```
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"`

The diagram shows the string "abcdefgh" with each character enclosed in a red box. A red double-headed arrow is positioned above the boxes for 'c', 'd', 'e', and 'f', indicating a slice operation. Below the string, two rows of indices are provided: the first row shows forward indices from 0 to 7, and the second row shows backward indices from -8 to -1.

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

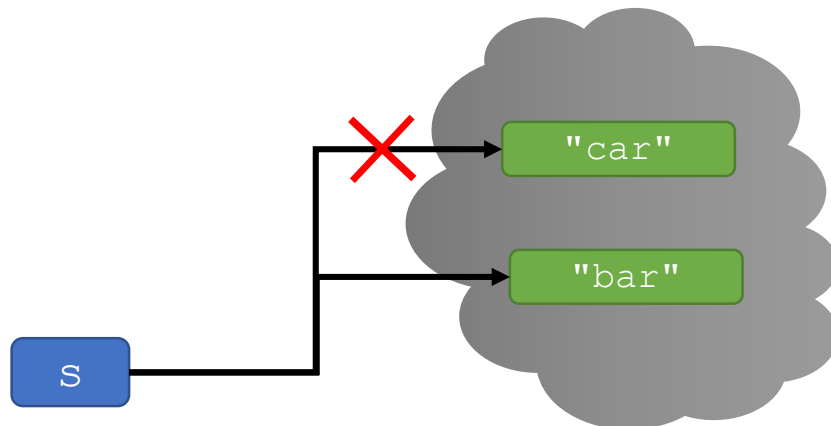
```
s = "car"
```

```
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!

INPUT/OUTPUT

PRINTING

- Used to **output** stuff to console

```
In [11]: 3+2
```

```
Out[11]: 5
```

- Command is `print`

```
In [12]: print(3+2)
```

```
5
```

- Printing many objects in the same command

- Separate objects using commas to output them separated by spaces

- Concatenate strings together using `+` to print as single object

- `a = "the"`

```
b = 3
```

```
c = "musketeers"
```

```
print(a, b, c)
```

```
print(a + str(b) + c)
```

*"Out" tells you it's an
interaction within the shell only*

*No "Out" means it is actually
shown to a user, apparent
when you edit/run files*

*Every piece being
concatenated must
be a string*

INPUT

- `x = input(s)`
 - Prints the value of the string `s`
 - User types in something and hits enter
 - That value is assigned to the variable `x`

- **Binds that value to a variable**

```
text = input("Type anything: ")  
  
print(5*text)
```

SHELL:

Type anything:

And it waits for
characters and
Enter to be hit

INPUT

- `x = input(s)`
 - Prints the value of the string `s`
 - User types in something and hits enter
 - That value is assigned to the variable `x`

- **Binds that value to a variable**

```
text = input("Type anything: ")
```

```
print(5*text)
```

"howdy"

SHELL:

Type anything: howdy

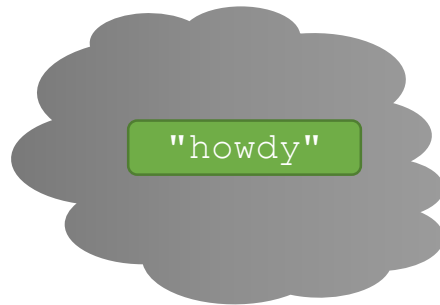
INPUT

- `x = input(s)`
 - Prints the value of the string `s`
 - User types in something and hits enter
 - That value is assigned to the variable `x`

- **Binds that value to a variable**

```
text = input("Type anything: ")
```

```
print(5*text)
```



SHELL:

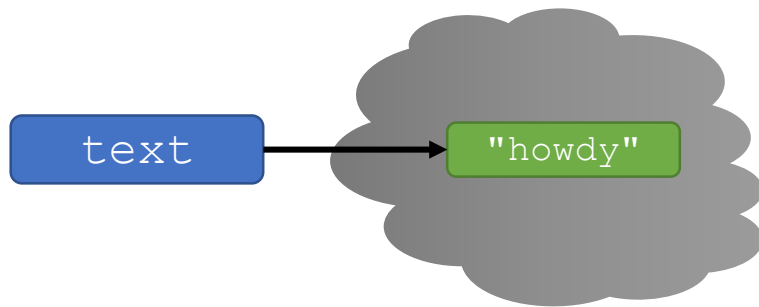
Type anything: howdy

INPUT

- `x = input(s)`
 - Prints the value of the string `s`
 - User types in something and hits enter
 - That value is assigned to the variable `x`
- **Binds that value to a variable**

```
text = input("Type anything: ")
```

```
print(5*text)
```



SHELL:

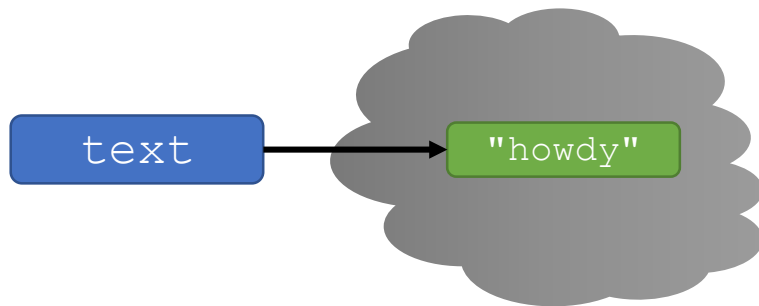
```
Type anything: howdy
```

INPUT

- `x = input(s)`
 - Prints the value of the string `s`
 - User types in something and hits enter
 - That value is assigned to the variable `x`
- **Binds that value to a variable**

```
text = input("Type anything: ")
```

```
print(5*text)
```



SHELL:

```
Type anything: howdy
howdyhowdyhowdyhowdyhowdy
```

INPUT

- `input` always returns an **str**, must cast if working with numbers

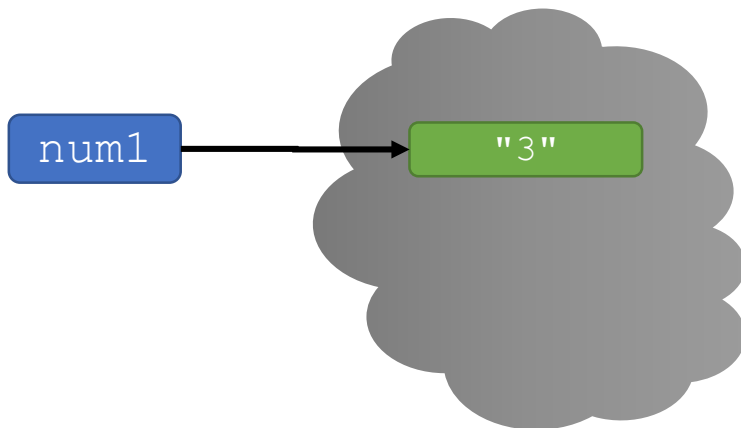
```
num1 = input("Type a number: ")
```

"3"

```
print(5*num1)
```

```
num2 = int(input("Type a number: "))
```

```
print(5*num2)
```



SHELL:

Type a number: 3

INPUT

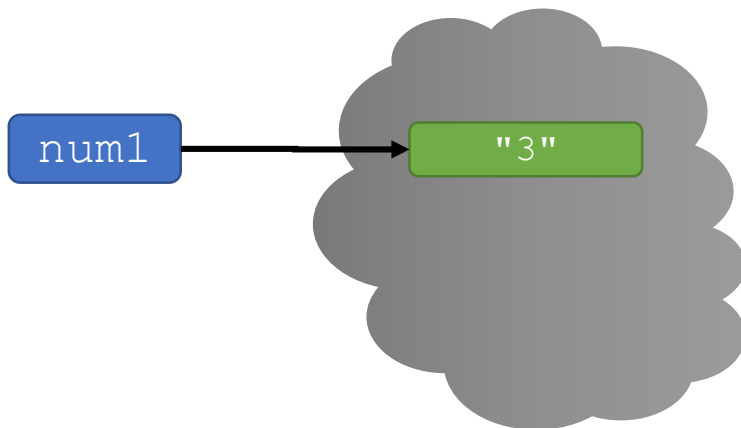
- `input` always returns an **str**, must cast if working with numbers

```
num1 = input("Type a number: ")
```

```
print(5*num1)
```

```
num2 = int(input("Type a number: "))
```

```
print(5*num2)
```



SHELL:

```
Type a number: 3
33333
```

INPUT

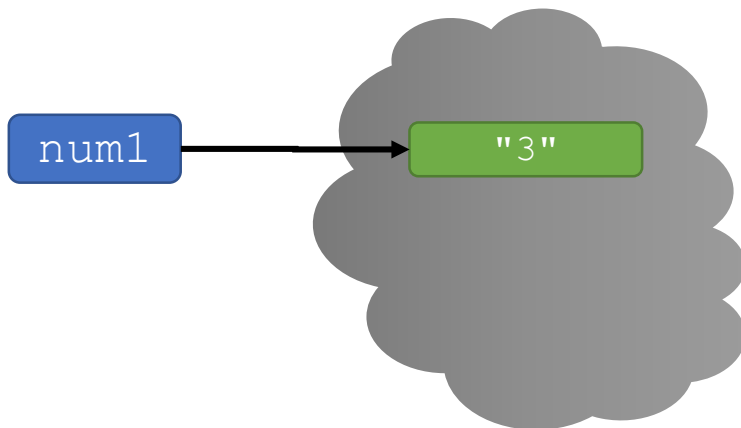
- `input` always returns an **str**, must cast if working with numbers

```
num1 = input("Type a number: ")
```

```
print(5*num1)
```

```
num2 = int(input("Type a number: "))
```

```
print(5*num2)
```



SHELL:

```
Type a number: 3
```

```
33333
```

```
Type a number: 3
```

INPUT

- `input` always returns an **str**, must cast if working with numbers

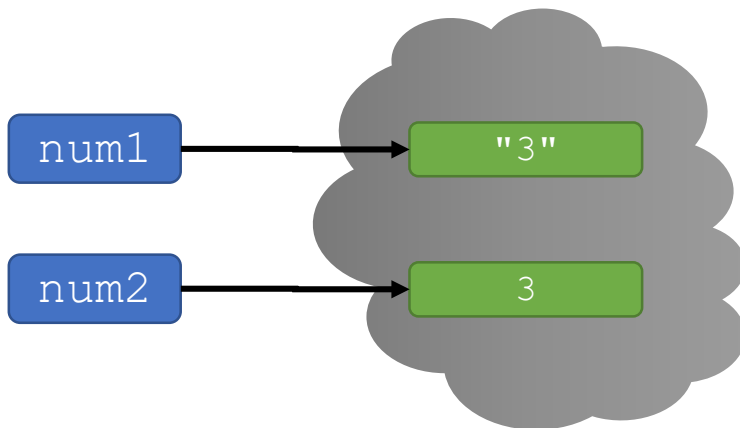
```
num1 = input("Type a number: ")
```

```
print(5*num1)
```

```
num2 = int(input("Type a number: "))
```

```
print(5*num2)
```

3



SHELL:

```
Type a number: 3
```

```
33333
```

```
Type a number: 3
```

INPUT

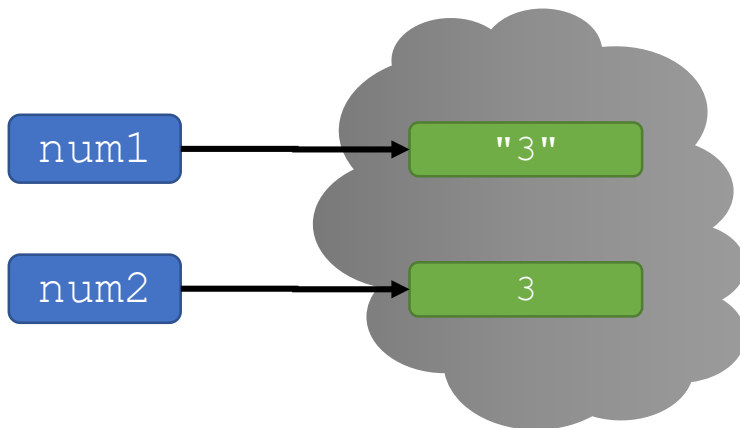
- `input` always returns an **str**, must cast if working with numbers

```
num1 = input("Type a number: ")
```

```
print(5*num1)
```

```
num2 = int(input("Type a number: "))
```

```
print(5*num2)
```



SHELL:

```
Type a number: 3
33333
Type a number: 3
15
```

YOU TRY IT!

- Write a program that
 - Asks the user for a verb
 - Prints “I can _ better than you” where you replace _ with the verb.
 - Then prints the verb 5 times in a row separated by spaces.
 - For example, if the user enters `run`, you print:

```
I can run better than you!  
run run run run run
```

AN IMPORTANT ALGORITHM: NEWTON'S METHOD

- Finds roots of a polynomial
 - E.g., find g such that $f(g, x) = g^3 - x = 0$
- Algorithm uses successive approximation
 - $\text{next_guess} = \text{guess} - \frac{f(\text{guess})}{f'(\text{guess})}$
- Partial code of algorithm that gets input and finds next guess

```
#Try Newton Raphson for cube root
x = int(input('What x to find the cube root of? '))
g = int(input('What guess to start with? '))
print('Current estimate cubed = ', g**3)

next_g = g - ((g**3 - x) / (3*g**2))
print('Next guess to try = ', next_g)
```

Handwritten red annotations:

- $f(g)$ above $(g**3 - x)$
- $f'(g)$ above $(3*g**2)$

F-STRINGS

- Available starting with Python 3.6
- Character `f` followed by a **formatted string literal**
 - Anything that can appear in a normal string literal
 - Expressions bracketed by curly braces `{ }`
- Expressions in curly braces evaluated at runtime, automatically converted to strings, and concatenated to the string preceding them

```
num = 3000
```

```
fraction = 1/3
```

```
print(num*fraction, 'is', fraction*100, '% of', num)
```

```
print(num*fraction, 'is', str(fraction*100) + '% of', num)
```

```
print(f'{num*fraction} is {fraction*100}% of {num}')
```

expressions

Introduces an extra space

BIG IDEA

Expressions can be
placed anywhere.

Python evaluates them!

CONDITIONS for BRANCHING

BINDING VARIABLES and VALUES

- In CS, there are two **notions of equal**
 - Assignment and Equality test
- `variable = value`
 - **Change the stored value** of variable to value
 - Nothing for us to solve, computer just does the action
- `some_expression == other_expression`
 - **A test for equality**
 - No binding is happening
 - Expressions are replaced by values and computer just does the comparison
 - Replaces the **entire line** with `True` or `False`

COMPARISON OPERATORS

- `i` and `j` are variable names
 - They can be of type ints, float, strings, etc.
- Comparisons below evaluate to the type **Boolean**
 - The Boolean type only has 2 values: `True` and `False`

`i > j`

`i >= j`

`i < j`

`i <= j`

*With strings, be careful
about case sensitivity:
'March' != 'march'*

`i == j` → **equality** test, `True` if `i` is the same as `j`

`i != j` → **inequality** test, `True` if `i` not the same as `j`

LOGICAL OPERATORS on bool

- a and b are variable names (with Boolean values)

`not a` → True if a is False
False if a is True

`a and b` → True if both are True

`a or b` → True if either or both are True

A	B	A and B	A or B
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

COMPARISON EXAMPLE

```
pset_time = 15
```

```
sleep_time = 8
```

```
print(sleep_time > pset_time)
```

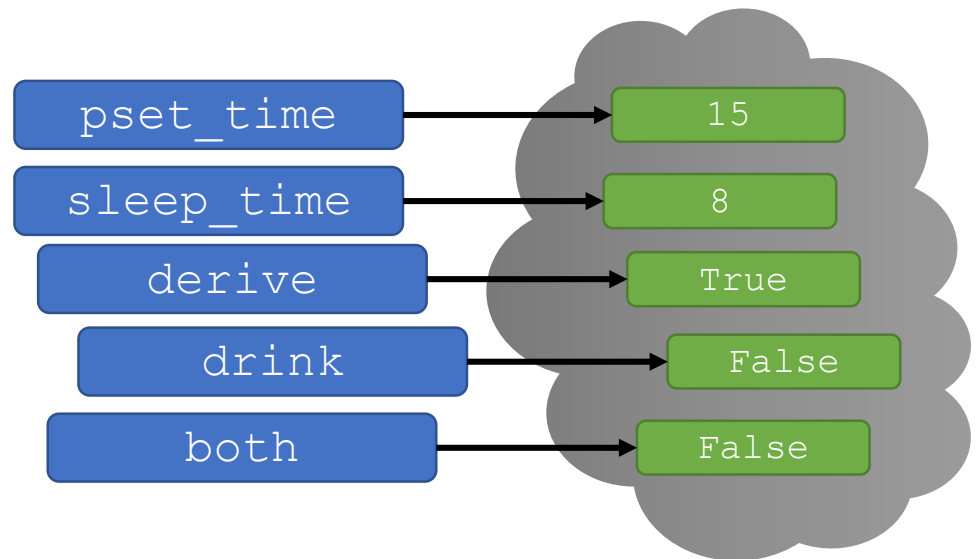
```
derive = True
```

```
drink = False
```

```
both = drink and derive
```

```
print(both)
```

*Prints the
boolean False*



*Prints the
boolean False*

YOU TRY IT!

- Write a program that
 - Saves a secret number in a variable.
 - Asks the user for a number guess.
 - Prints a bool `False` or `True` depending on whether the guess matches the secret.

WHY bool?

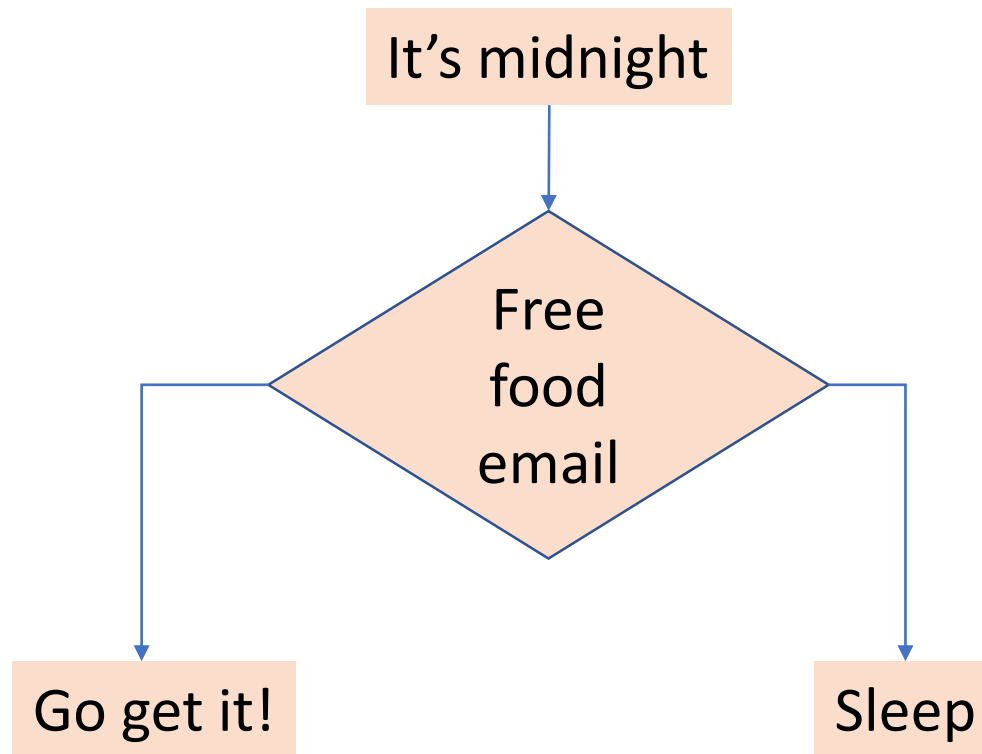
- When we get to flow of control, i.e. branching to different expressions based on values, we need a way of knowing if a condition is true
- E.g., if something is true, do this, otherwise do that

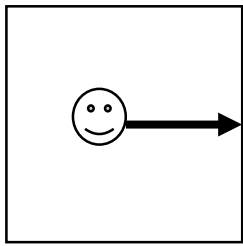
Boolean

Some
commands

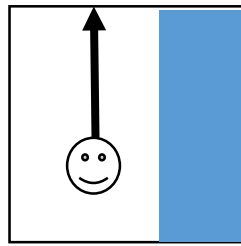
Some other
commands

INTERESTING ALGORITHMS INVOLVE DECISIONS

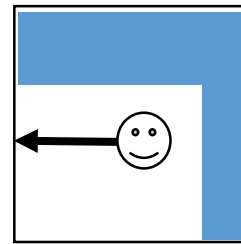




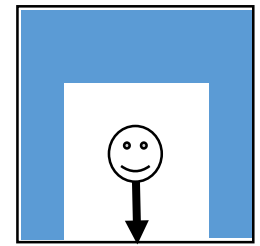
If right clear,
go right



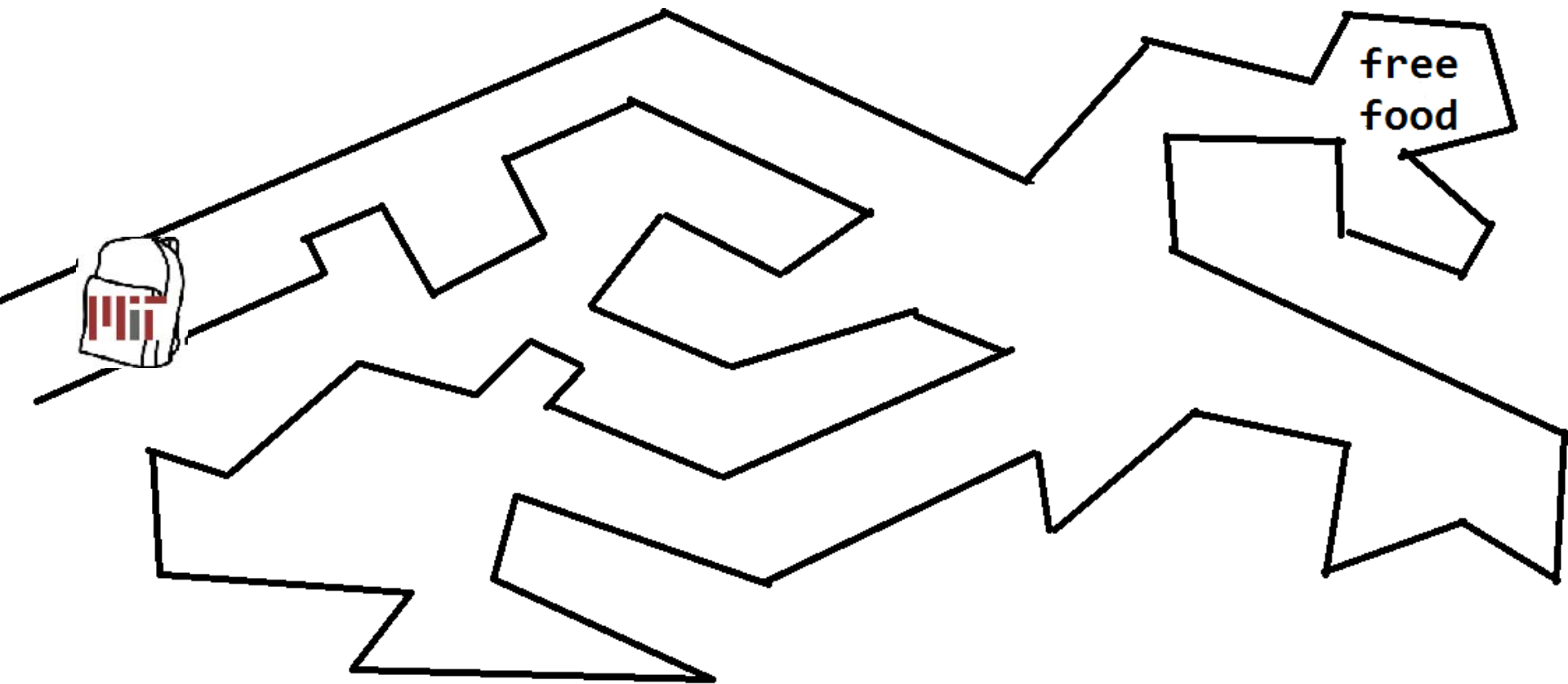
If right blocked,
go forward



If right and
front blocked,
go left



If right , front,
left blocked,
go back



BRANCHING IN PYTHON

```
if <condition>:  
    <code>  
    <code>  
    ...  
<rest of program>
```

- <condition> has a value `True` or `False`
- **Indentation matters** in Python!
- Do code within if block if condition is `True`

BRANCHING IN PYTHON

```
if <condition>:  
    <code>  
    <code>  
    ...  
<rest of program>
```

```
if <condition>:  
    <code>  
    <code>  
    ...  
else:  
    <code>  
    <code>  
    ...  
<rest of program>
```

- <condition> has a value `True` or `False`
- **Indentation matters** in Python!
- Do code within if block when condition is `True` **or** code within else block when condition is `False`.

BRANCHING IN PYTHON

```
if <condition>:  
    <code>  
    <code>  
    ...  
<rest of program>
```

```
if <condition>:  
    <code>  
    <code>  
    ...  
else:  
    <code>  
    <code>  
    ...  
<rest of program>
```

```
if <condition>:  
    <code>  
    <code>  
    ...  
elif <condition>:  
    <code>  
    <code>  
    ...  
elif <condition>:  
    <code>  
    <code>  
    ...  
<rest of program>
```

- <condition> has a value True or False
- **Indentation matters** in Python!
- Run the **first block** whose corresponding <condition> is True

BRANCHING IN PYTHON

```
if <condition>:  
    <code>  
    <code>  
    ...  
<rest of program>
```

```
if <condition>:  
    <code>  
    <code>  
    ...  
else:  
    <code>  
    <code>  
    ...  
<rest of program>
```

```
if <condition>:  
    <code>  
    <code>  
    ...  
elif <condition>:  
    <code>  
    <code>  
    ...  
elif <condition>:  
    <code>  
    <code>  
    ...  
<rest of program>
```

```
if <condition>:  
    <code>  
    <code>  
    ...  
elif <condition>:  
    <code>  
    <code>  
    ...  
else:  
    <code>  
    <code>  
    ...  
<rest of program>
```

- <condition> has a value True or False
- **Indentation matters** in Python!
- Run the **first block** whose corresponding <condition> is True. The else block runs when no conditions were True

BRANCHING EXAMPLE

```
pset_time = ???
```

```
sleep_time = ???
```

```
if (pset_time + sleep_time) > 24:
```

```
    print("impossible!")
```

```
elif (pset_time + sleep_time) >= 24:
```

```
    print("full schedule!")
```

```
else:
```

```
    leftover = abs(24-pset_time-sleep_time)
```

```
    print(leftover, "h of free time!")
```

```
print("end of day")
```

Condition that evaluates to a Boolean

This indented code executed
if line above is True

This indented code executed
if line above is True and the if
condition is False

This else block runs only
if previous conditions
were all False

YOU TRY IT!

- Semantic structure matches visual structure
- Fix this buggy code (hint, it has bad indentation)!

```
x = int(input("Enter a number for x: "))
y = int(input("Enter a different number for y: "))
if x == y:
    print(x, "is the same as", y)
print("These are equal!")
```

INDENTATION and NESTED BRANCHING

- Matters in Python
- How you **denote blocks of code**

```
x = float(input("Enter a number for x: "))
y = float(input("Enter a number for y: "))
if x == y:
    print("x and y are equal")
    if y != 0:
        print("therefore, x / y is", x/y)
elif x < y:
    print("x is smaller")
else:
    print("y is smaller")
print("thanks!")
```

5	5	0
5	0	0
True	False	True
<-		<-
True		False
<-		
	False	
	<-	
<-	<-	<-

BIG IDEA

Practice will help you
build a mental model of
how to trace the code

Indentation does a lot of the work for you!

YOU TRY IT!

- What does this code print with
 - `y = 2`
 - `y = 20`
 - `y = 11`
- What if `if x <= y:` becomes `elif x <= y: ?`

```
answer = ''
x = 11
if x == y:
    answer = answer + 'M'
if x >= y:
    answer = answer + 'i'
else:
    answer = answer + 'T'
print(answer)
```

YOU TRY IT!

- Write a program that
 - Saves a secret number.
 - Asks the user for a number guess.
 - Prints whether the guess is too low, too high, or the same as the secret.

BIG IDEA

Debug early,
debug often.

Write a little and test a little.

Don't write a complete program at once. It introduces too many errors.

Use the Python Tutor to step through code when you see something unexpected!

SUMMARY

- Strings provide a new data type
 - They are **sequences of characters**, the **first one at index 0**
 - They can be indexed and sliced
- Input
 - Done with the **input** command
 - Anything the user inputs is **read as a string object!**
- Output
 - Is done with the **print** command
 - Only objects that are printed in a .py code file will be **visible in the shell**
- Branching
 - Programs execute **code blocks** when conditions are true
 - In an `if-elif-elif...` structure, the **first condition that is True** will be executed
 - **Indentation matters** in Python!