

# Welcome to 6.00.1X

---

# OVERVIEW OF COURSE

---

- learn computational modes of thinking
- master the art of computational problem solving
- make computers do what you want them to do



<https://onthehumanityblog.files.wordpress.com/2014/09/computerthink.gif>

# TOPICS

---

- represent knowledge with **data structures**
- **iteration and recursion** as computational metaphors
- **abstraction** of procedures and data types
- **organize and modularize** systems using object classes and methods
- different classes of **algorithms**, searching and sorting
- **complexity** of algorithms

# WHAT DOES A COMPUTER DO

---

- Fundamentally:
  - performs **calculations**  
a billion calculations per second!  
two operations in same time light travels 1 foot
  - **remembers** results  
100s of gigabytes of storage!  
typical machine could hold 1.5M books of standard size
- What kinds of calculations?
  - **built-in** to the language
  - ones that **you define** as the programmer

# SIMPLE CALCULATIONS ENOUGH?

---

- Searching the World Wide Web
  - 45B pages; 1000 words/page; 10 operations/word to find
  - Need 5.2 days to find something using simple operations
- Playing chess
  - Average of 35 moves/setting; look ahead 6 moves; 1.8B boards to check; 100 operations/choice
  - 30 minutes to decide each move
- Good algorithm design also needed to accomplish a task!

# ENOUGH STORAGE?

---

- What if we could just pre-compute information and then look up the answer
  - Playing chess as an example
  - Experts suggest  $10^{123}$  different possible games
  - Only  $10^{80}$  atoms in the observable universe

# ARE THERE LIMITS?

---

- Despite its speed and size, a computer does have limitations
  - Some problems still too complex
    - Accurate weather prediction at a local scale
    - Cracking encryption schemes
  - Some problems are fundamentally impossible to compute
    - Predicting whether a piece of code will always halt with an answer for any input





# TYPES OF KNOWLEDGE

---

- computers know what you tell them
- **declarative knowledge** is **statements of fact**.
  - there is candy taped to the underside of one chair
- **imperative knowledge** is a **recipe** or “how-to” knowledge
  - 1) face the students at the front of the room
  - 2) count up 3 rows
  - 3) start from the middle section’s left side
  - 4) count to the right 1 chair
  - 5) reach under chair and find it

what  
imperative  
how.

# A NUMERICAL EXAMPLE

what

- square root of a number  $x$  is  $y$  such that  $y^*y = x$
- recipe for deducing square root of number  $x$  (e.g. 16)
  - 1) Start with a **guess**,  $q$
  - 2) If  $q^*q$  is **close enough** to  $x$ , stop and say  $q$  is the answer
  - 3) Otherwise make a **new guess** by averaging  $q$  and  $x/q$
  - 4) Using the new guess, **repeat** process until close enough

↓  
few.

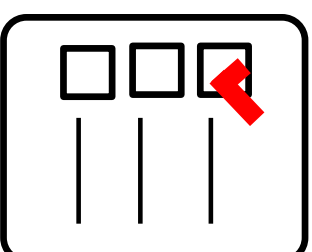
$q$	$q^*q$	$x/q$	$(q+x/q)/2$
3	9	5.333	4.1667
4.1667	17.36	3.837	4.0035
4.0035	16.0277	3.997	4.000002

$\Rightarrow q_{\text{new}} = (q + x/q)/2$   
Avg of  $q$  and  $x/q$

# WHAT IS A RECIPE

---

- 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**

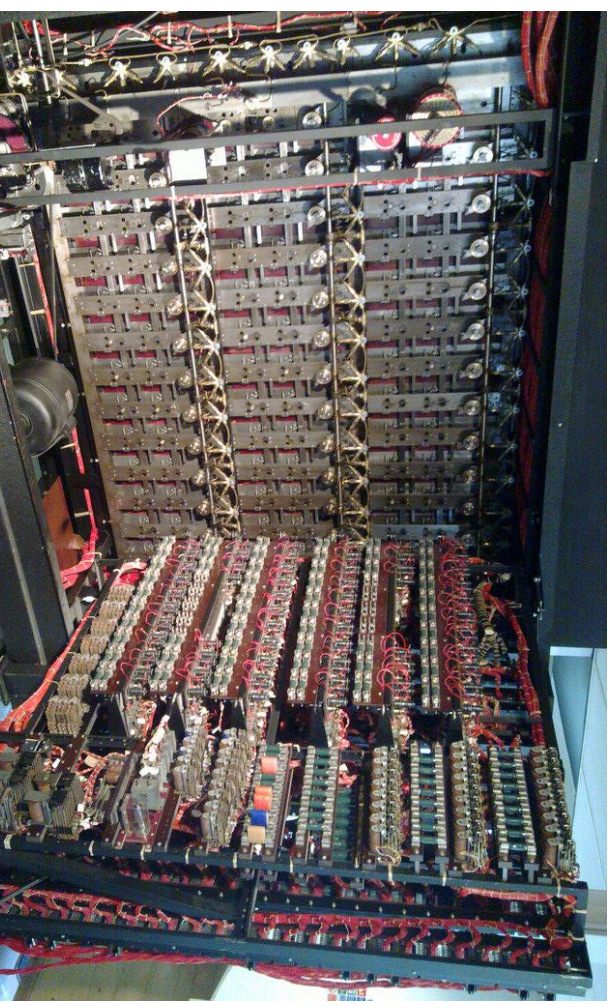


Steps 1+2+3 = an **algorithm**!

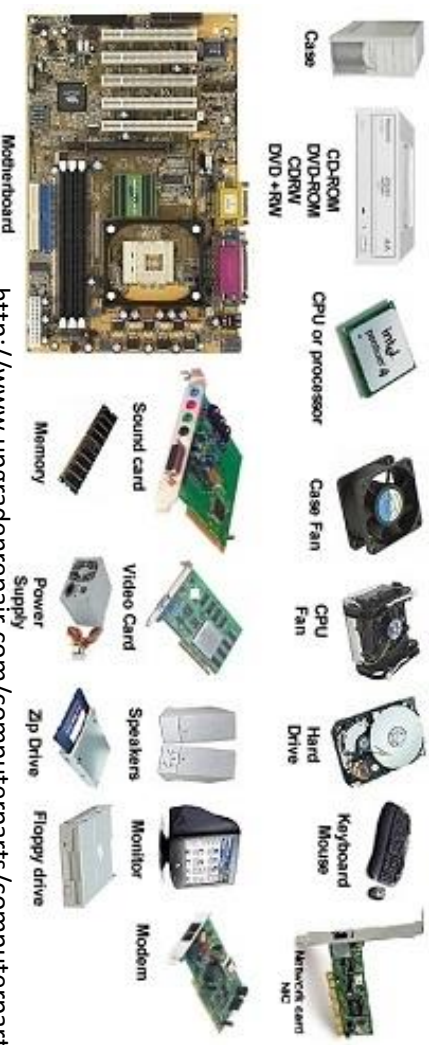
---

# COMPUTERS ARE MACHINES

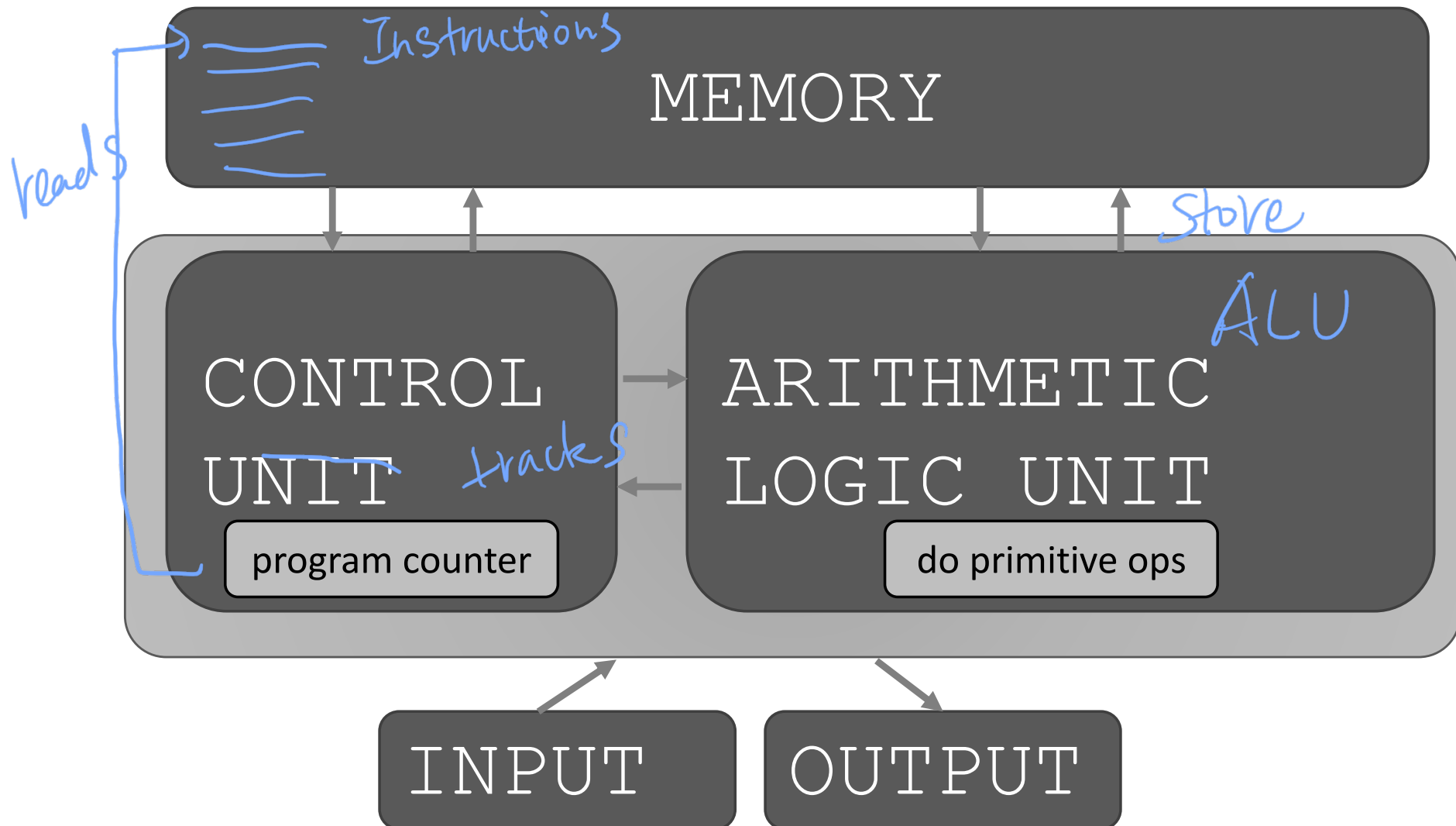
- how to capture a recipe in a mechanical process
- **fixed program** computer
  - calculator
  - Alan Turing's Bombe
- **stored program** computer
  - machine stores and executes instructions



CC-BY SA 2.0 dlapper



# BASIC MACHINE ARCHITECTURE



# STORED PROGRAM COMPUTER

---

- sequence of **instructions stored** inside computer
  - built from predefined set of primitive instructions
    - 1) arithmetic and logic
    - 2) simple tests
    - 3) moving data
- special program (interpreter) **executes each instruction in order**
  - use tests to change flow of control through sequence
  - stop when done



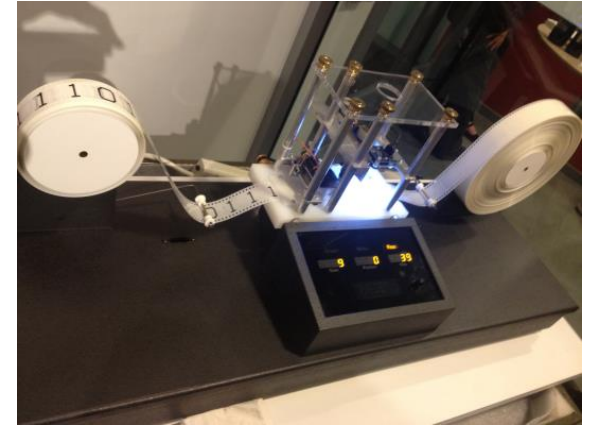
# BASIC PRIMITIVES



- Turing showed you can **compute anything** using 6 primitives

← → Scan read N/A.

- modern programming languages have more convenient set of primitives
- can abstract methods to **create new primitives**
- anything computable in one language is computable in any other programming language



By Gabrielf (Own work) [CC BY-SA 3.0  
(<http://creativecommons.org/licenses/by-sa/3.0/>)], via  
Wikimedia Commons



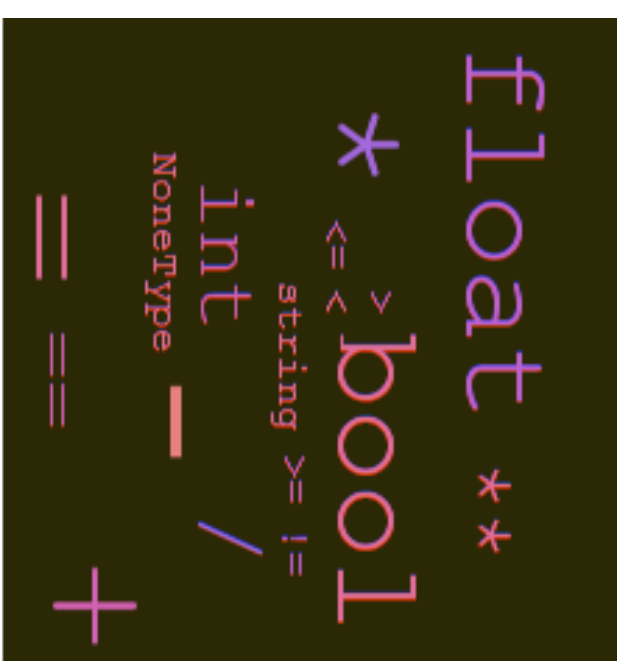


- 
- fixed program vs stored program
  - Machine Architecture      ↳ interpreter

# CREATING RECIPES

---

- a programming language provides a set of primitive **operations**
- **expressions** are complex but legal combinations of primitives in a programming language
- expressions and computations have **values** and meanings in a programming language



# ASPECTS OF LANGUAGE

---

- **syntax** *syntact.*
  - English: "cat dog boy" → not syntactically valid  
"cat hugs boy" → syntactically valid
  - programming language: "hi" 5 → not syntactically valid  
3.2 \* 5 → syntactically valid

# ASPECTS OF LANGUAGES

---

- **static semantics** is which syntactically valid strings have meaning
  - English: "I are hungry" → syntactically valid but static semantic error
  - programming language:  $3 \cdot 2 * 5$  → syntactically valid
  - $3 + "hi"$  → static semantic error

# ASPECTS OF LANGUAGES

---

■ **semantics** is the meaning associated with a syntactically correct string of symbols with no static semantic errors

“语义正确”

- English: can have many meanings –
  - “Flying planes can be dangerous”
  - “This reading lamp hasn’t uttered a word since I bought it?”
- programming languages: have only one meaning but may not be what programmer intended

# WHERE THINGS GO WRONG

---

- **syntactic errors** 语法错误
  - common and easily caught
- **static semantic errors** 静态语义错误
  - some languages check for these before running program
  - can cause unpredictable behavior
- no semantic errors but **different meaning than what programmer intended** 语义不唯一
  - program crashes, stops running
  - program runs forever
  - program gives an answer but different than expected

# OUR GOAL

---

- Learn the syntax and semantics of a programming language
- Learn how to use those elements to translate “recipes” for solving a problem into a form that the computer can use to do the work for us
- Learn computational modes of thought to enable us to leverage a suite of methods to solve complex problems



---

# PYTHON PROGRAMS

---

- a **program** is a sequence of definitions and commands
  - definitions **evaluated** *what*
  - commands **executed** by Python interpreter in a shell *what to do,*
- **commands** (statements) instruct interpreter to do something *落口, 落口, 落口,*
- can be typed directly in a **shell** or stored in a **file** that is read into the shell and evaluated

In python, all objects live on heap.


# OBJECTS

---

- programs manipulate **data objects**
- objects have a **type** that defines the kinds of things programs can do to them
- objects are *consist of*:
  - scalar (cannot be subdivided)
  - **non-scalar** (have internal structure that can be accessed)

# SCALAR OBJECTS

---

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

```
In [1]: type(5)  
Out[1]: int
```

*What you  
write into the  
Python shell  
what shows after  
hitting enter*

```
In [2]: type(3.0)  
Out[2]: float
```

# TYPE CONVERSIONS (CAST)

---

- can **convert object of one type to another**
- `float(3)` converts integer 3 to float 3.0
- `int(3.9)` truncates float 3.9 to integer 3

# PRINTING TO CONSOLE

---

- To show output from code to a user, use `print` command

In [11]: 3+2

Out[11]: 5

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

5

*no 'Out' because no value  
returned, just something printed*

# EXPRESSIONS

---

- **combine objects and operators** to form expressions
  - an expression has a **value**, which has a type
  - syntax for a simple expression
- `<object> <operator> <object>`

# OPERATORS ON ints and floats

- $i + j$  → the **sum**
  - if both are ints, result is int
- $i - j$  → the **difference**
  - if either or both are floats, result is float
- $i * j$  → the **product**
- $i / j$  → **division** ——— - result is float
- $i // j$  → **int division** ——— - result is int, quotient without remainder
- $i \% j$  → the **remainder** when  $i$  is divided by  $j$
- $i ** j$  →  $i$  to the **power** of  $j$

整数



# SIMPLE OPERATIONS

---

- parentheses used to tell Python to do these operations first
  - $3*5+1$  evaluates to 16
  - $3*(5+1)$  evaluates to 18
- **operator precedence** without parentheses
  - $**$
  - $*$
  - $/$
  - $+$  and  $-$  executed left to right, as appear in expression

---

# BINDING VARIABLES AND VALUES

---

- equal sign is an **assignment** of a value to a variable name

*variable*

`pi` =

`3.14159`

*value*

`pi_approx` =

`22 / 7`

*If use `22 / 7`, value of expression is 3*

- value stored in computer memory
- an assignment binds name to value
- retrieve value associated with name or variable by invoking the name, by typing `pi`

# ABSTRACTING EXPRESSIONS

---

- why **give names** to values of expressions?
- **reuse names** instead of values
- easier to change code later

```
pi = 3.14159  
radius = 2.2  
area = pi * (radius**2)
```

# PROGRAMMING vs MATH

---

- in programming, you do not “solve for x”

```
pi = 3.14159
radius = 2.2
# area of circle
area = pi * (radius**2)
radius = radius+1
```

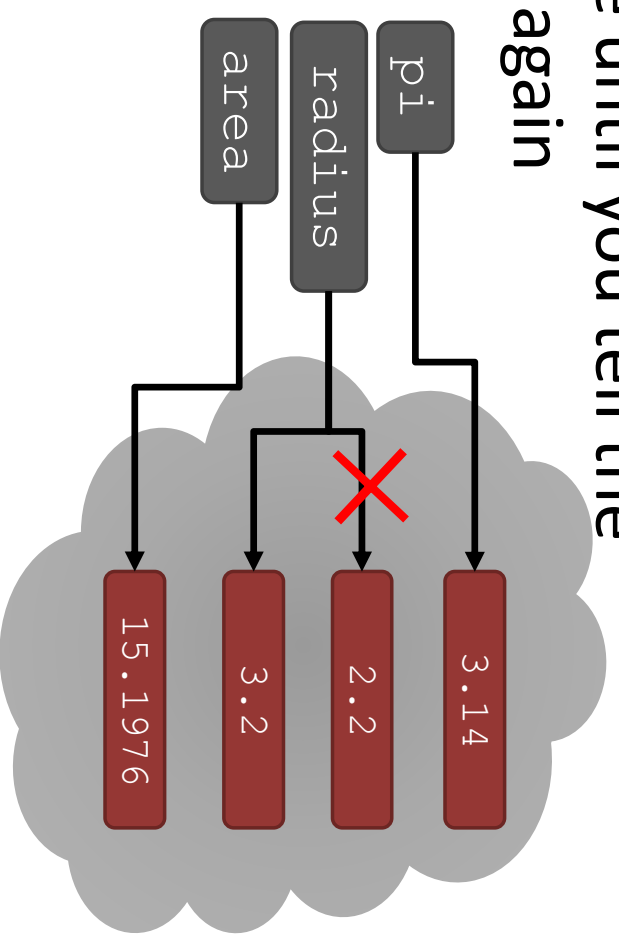
an assignment  
- value on the right  
- name on the left  
- equivalent is  $radius += 1$

# CHANGING 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
```





# COMPARISON OPERATORS ON `int` and `float`

---

- `i` and `j` are any variable names

`i > j`

`i >= j`

`i < j`

`i <= j`

`i == j` → **equality** test, `True` if `i` equals `j`

`i != j` → **inequality** test, `True` if `i` not equal to `j`



# LOGIC OPERATORS ON bools

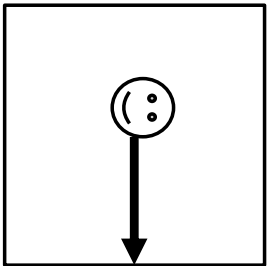
---

- **a and b are any variable names**

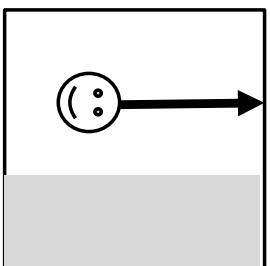
not a      $\rightarrow$  True if a is False  
             False if a is True

a and b    $\rightarrow$  True if both are True

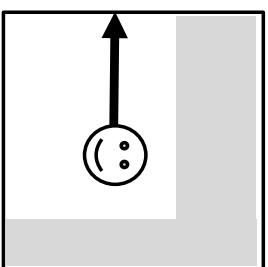
a or b     $\rightarrow$  True if either or both are True



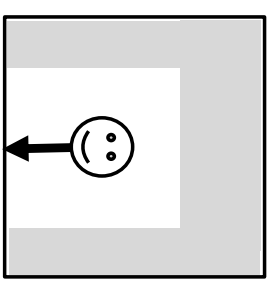
If right clear,  
go right



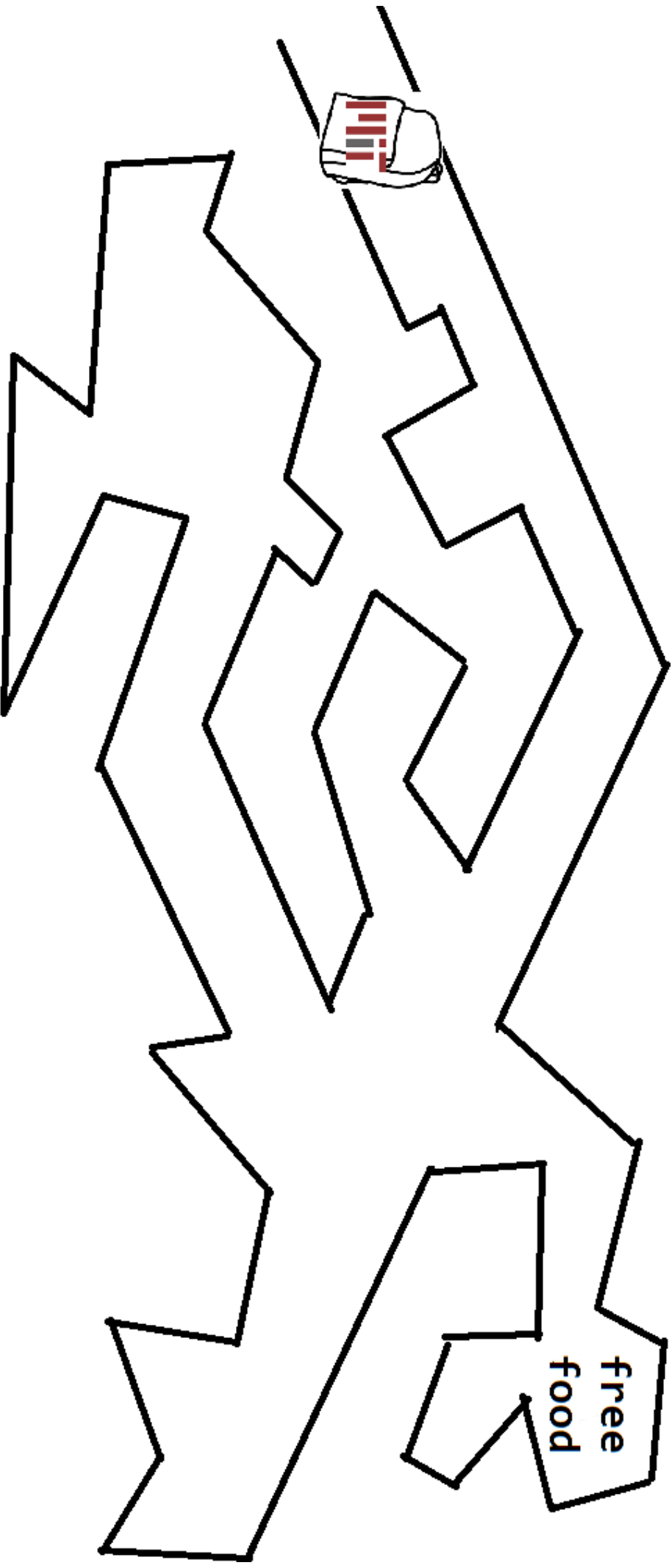
If right blocked,  
go forward



If right and  
front blocked,  
go left



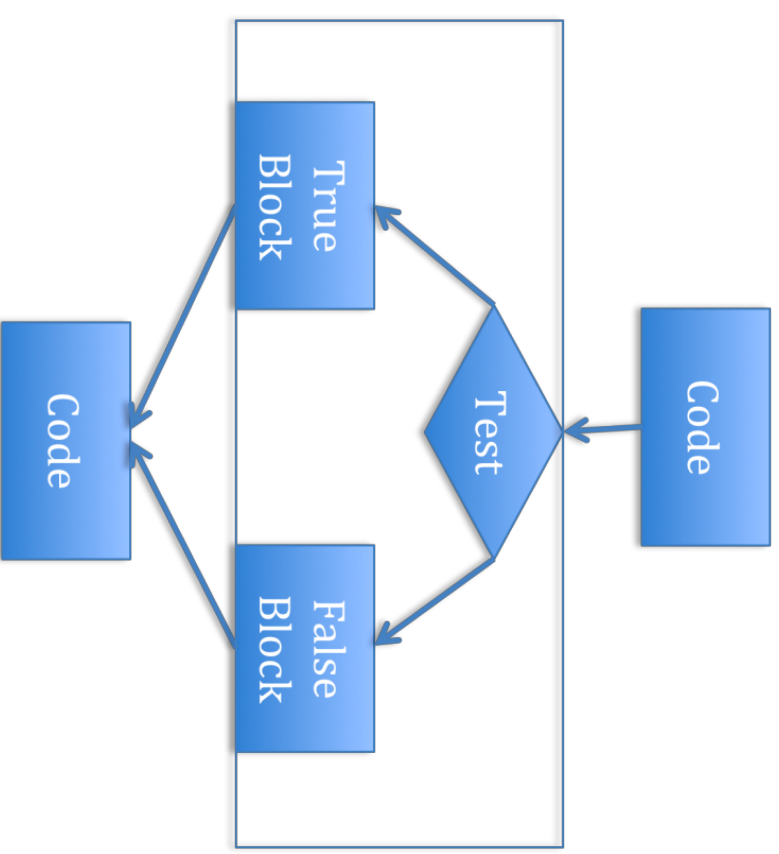
If right, front,  
left blocked,  
go back



# BRANCHING PROGRAMS

---

- The simplest branching statement is a **conditional**
  - A test (expression that evaluates to `True` or `False`)
  - A block of code to execute if the test is `True`
  - An optional block of code to execute if the test is `False`



# A SIMPLE EXAMPLE

---

```
x = int(input('Enter an integer: '))
if x%2 == 0:
    print('\n')
    print('Even')
else:
    print('\n')
    print('Odd')
print('Done with conditional')
```

# SOME OBSERVATIONS

---

- The expression `x%2 == 0` evaluates to **True** when the remainder of `x` divided by 2 is 0
- Note that `==` is used for comparison, since `=` is reserved for assignment
- The indentation is important – each indented set of expressions denotes a block of instructions
  - For example, if the last statement were indented, it would be executed as part of the `else` block of code
- Note how this indentation provides a visual structure that reflects the semantic structure of the program

# NESTED CONDITIONALS

---

```
if x%2 == 0:
    if x%3 == 0:
        print('Divisible by 2 and 3')
    else:
        print('Divisible by 2 and not by 3')
elif x%3 == 0:
    print('Divisible by 3 and not by 2')
```

# COMPOUND BOOLEANS

---

```
if x < y and x < z:  
    print('x is least')  
elif y < z:  
    print('y is least')  
else:  
    print('z is least')
```

# CONTROL FLOW - BRANCHING

---

```
if <condition>:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

```
if <condition>:  
    <expression>  
    <expression>  
    ...  
elif <condition>:  
    <expression>  
    <expression>  
    ...  
else:  
    <expression>  
    <expression>  
    ...
```

- `<condition>` has a value `True` or `False`
- evaluate expressions in that block if `<condition>` is `True`



# INDENTATION

---

- 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!")
```

# = VS ==

---

```
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!")
```

What if `x = y` here?  
get a `SyntaxError`

# WHAT HAVE WE ADDED?

---

- Branching programs allow us to make choices and do different things
- But still the case that at most, each statement gets executed once.
- So maximum time to run the program depends only on the length of the program
- These programs run in **constant time**