What is a program?

- It consists of two components:
 - Data (numbers, characters, true/false)

 \circ Steps

- A program goes through a number of steps with pieces of data to achieve a result:
 - $\,\circ\,$ Printing text to screen
 - \circ Collecting information
 - Performing calculations
- Example: Long Division

Programming Languages

- Computer programmers write programs for computers using one or more programming languages
- Some languages are better for one type of program or one style of user interface than for others
- You may have heard of some programming languages: Basic, Lisp, C/C++, Java, Python, Assembly Language, and Others

"Hello, World" Versions

• Java:

```
public class Hello {
```

```
public static void main(String[] args) {
  System.out.println("Hello World");
}
```

- Basic: 10 PRINT "HELLO WORLD"
- Fortran: program Helloworld 10 FORMAT (1X,11HHELLO WORLD) WRITE(6,10) END "HELLO WORLD"
- Python: print ("Hello World")

```
• C:
```

```
#include <stdio.h>
 #include <stdlib.h>
 int main(void)
 ł
 printf("Hello, world\n");
  return EXIT SUCCESS;
 }
• Scheme:
(display "Hello, World!")
(newline)
```

Programming Languages

•A *programming language* specifies the words and symbols that we can use to write a program

•A programming language employs a set of rules that dictate how the words and symbols can be put together to form valid *program statements*

•A programming language has both *syntax* and *semantics*

Syntax and Semantics

- The *syntax rules* of a language define <u>how we can put together</u> symbols, reserved words, and identifiers to make a valid program
- The semantics of a program statement define what that statement means (its purpose or role in a program)
- A program that is syntactically correct is not necessarily logically (semantically) correct
- A program will always do what we <u>tell</u> it to do, not what we <u>meant</u> to tell it to do

Program Structure

- In a programming language:
 - A program is made up of one or more instructions, or statements, which perform operations upon various pieces of data
 - Data may be stored in variables
 - Related groups of statements may be organized into *methods*
 - Related variables and methods may be organized into larger units, such as *classes* and *modules*

Basic Definitions

- **Statement:** A piece of code representing a complete step in a program
- **Variable:** A named space in program memory for storing a piece of data.
- *Method*: A named set of instructions that acts upon supplied data in order to accomplish some goal
- *Module, Library, etc.*: A body of pre-written code that you can incorporate into a program
- **Class:** A way of organizing variables and methods, usually for modeling a real-life entity

<u>White Space</u>

- Spaces, blank lines, and tabs are called *white space*
- White space is used to separate words and symbols in a program. Extra white space is usually ignored, depending on the language
- A valid program can be formatted many ways
- Programs should be formatted to enhance readability, using consistent indentation
- In some programming languages, like Python, correct use of indentation is necessary in order to indicate organization of code, as we will see soon.

<u>Printing</u>

- One of the most basic steps in a simple CLI-based program is printing text to the screen
- There are a number of variations on this step, some simpler and some more complex
 - Single line of text vs. multiple lines
 - Printing with a terminal newline vs. without
 - Printing plain strings of text
 - o ... or other data types
 - ... or the results of expressions

Variable Declaration

- A *variable* is a name for a location in memory
- A variable must be *declared* by specifying its <u>name</u> and <u>its initial value</u> (*example below uses <u>Python</u>*)

• In some languages (*e.g., Java*), variables are of a specific type, but Python is more flexible

Value Assignment

- An *assignment statement* gives the variable an actual value in memory
- The equals sign provides this function

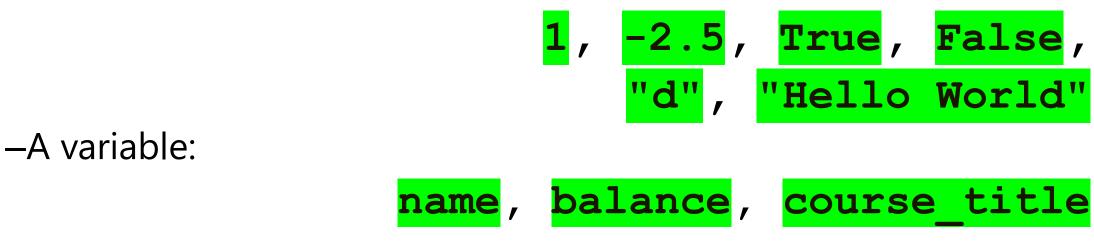
• The expression on the right is <u>evaluated</u> and the result is <u>stored</u> as the value of the variable on the left

total = 55

- Any value previously stored in **total** is overwritten
- Some languages like Java will restricted the kinds of values you can assign to a variable, based on its type

Operators and Operands

- <u>Operand</u>: Can be any element that has some value:
 - -A literal:



student.get name()

-The result of a method call:

Operators and Operands

<u>Operator</u>: Something that *computes a result* using one or more operands:

1 🕂 2 6 🕖 3 (!)ightIsOn count (+=) 1 5 🛛 4 😑 10 🗶 2 18 - 6 - 18

Expressions

- An *expression* is a combination of one or more <u>operators</u> and <u>operands</u>
- *Arithmetic expressions* compute numeric results and make use of the arithmetic operators:

Add	+	Integer	11
Subtract	-	(floor)	
Multiply	*	Division	
Divide	1		
Remainder	90	Exponent	**

• If either or both operands used by an arithmetic operator are floating point (i.e., **decimal**), then the result is a floating point

Operator Precedence

• Operands and operators can be combined into **complex expressions**

result = total + count / maxi - offset

- Operators have a well-defined precedence which determines the order in which they are evaluated
- Multiplication, division, and remainder are evaluated prior to addition, subtraction, and string concatenation
- Arithmetic operators with the same precedence are evaluated from left to right, but **parentheses** can be used to **force the evaluation order**
- In fact, arithmetic expressions can be combined with other operators to create boolean expressions....

Boolean Expressions

- A <u>boolean expression</u> is one that returns either of two possible values: <u>True</u> or <u>False</u>
- Boolean expressions, like arithmetic ones, use operators, such as the following <u>equality</u> and <u>relational</u> operators:
 - == equal to
 - ! = not equal to
 - < less than
 - > greater than
 - <= less than or equal to
 - >= greater than or equal to
- These address questions of *ordering*, where things can be consider greater/lesser, coming before/after, etc.

Boolean (relational) Expressions

5 < 7
7 >= 5
x == 98
offer < minimum_bid
grade+1 >= a_grade
t weight < weight</pre>

Logical Operators

• The following *logical operators* can also be used in boolean expressions:



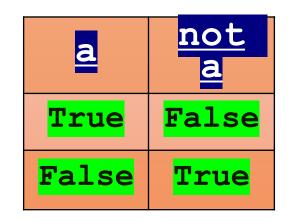
 They operate on <u>boolean operands</u> and produce <u>boolean</u> results: **True** or **False**

-Logical **NOT** is a **unary** operator => **one operand**

-AND and OR are **binary** operators => **two operands**

Logical NOT

- The *logical NOT* operation is also called *logical negation* or *logical complement*
- If some boolean condition a is True, then not a is
 False
- If a is False, then not a is True
- Logical operations can be shown with a *truth table*



Logical AND and Logical OR

• The *logical AND* expression

is True if both a and b are True, and False otherwise

and b

- The *logical OR* expression
- a or b
 is True if <u>at least</u> one of a or b is True, and False otherwise

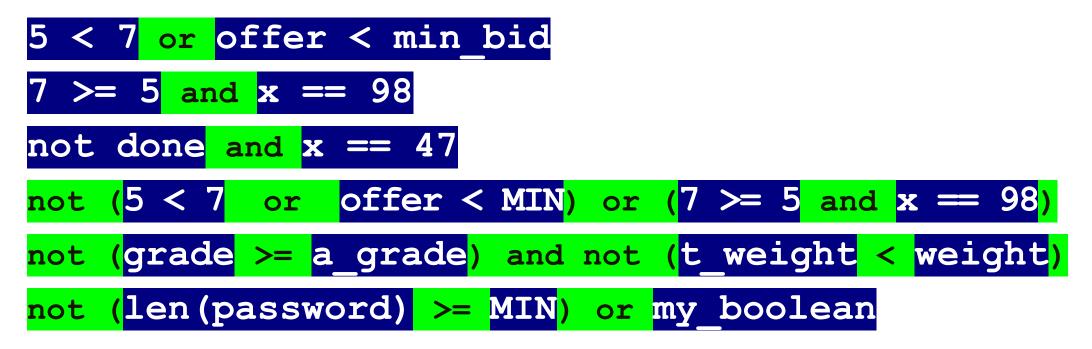
Logical Operators

- A <u>truth table</u> shows all possible True False combinations of the terms
- Since and and or each have two operands, there are four possible combinations of conditions a and b

a	b	a and b	a or b
True	True	True	True
True	False	False	True
False	True	False	True
False	False	False	False

More Boolean Expressions

 NOTE: You should look at these <u>primarily</u> as examples of how boolean expressions can be combined into more complex ones. (<u>Python</u> style below!)



Reading Input

- Programs generally need input on which to operate •
- Specific languages have ways that allow us to get this information from the user, when writing a command-line application
- It can also be used to <u>halt program execution</u> until the user presses **Enter**
- To use it, you will need:

1) The method, code, etc. that gets user input

2) Prompt text (e.g., "Please type your name:

Reading Input

- The input method will:
 - 1) Print your specified prompt text
 - 2) Wait for the user to press Enter
 - 3) Return the user's input as some type of data -- often a <u>string</u> (an empty string, if the user entered no text)
- To halt program execution, you can use input prompts <u>without</u> <u>storing the result.</u>
- This can be useful when you want the program to <u>stop</u> at certain points

Interactive Applications (CLI)

- An interactive program with a command line interface contains a sequence of steps to:
 - Prompt the user
 - Get the user's responses
 - Process the data as input is received (or after)
- Python example:

name = input("Enter name: ")
age = int(input("Enter age: "))
money = float(input("Money: \$"))

Flow of Control

- Default order of statement execution is linear: one after another in sequence
- But, sometimes we need to decide <u>which</u> statements to execute and/or <u>how many times</u>
- These decisions are based on *boolean expressions* (or "conditions") that evaluate to **True** or **False**
- The resulting order of statement execution, according to these decisions, is called the *flow of control*

Flow of Control

- We can speak of three forms of flow control:
 - *1. Sequencing* is the most basic form of flow control: the mere execution of steps, in order, one after the other.
 - 2. Branching involves a choice between one or more potential options for which statement(s) to execute next, before continuing
 - *3. Repetition* involves executing a block of code, over and over, until reaching a logical stopping point
- By combining these basic types of flow control, you can forge increasingly complex and sophisticated programs!

<u>Branching - Conditional Structures</u>

- A <u>conditional structure</u> decides which program statement(s) will be executed next
- We use boolean conditions to make basic decisions as the program runs.
- Recall the quadratic formula example:

• Check if
$$a = 0$$
, if $b = 0$, etc.

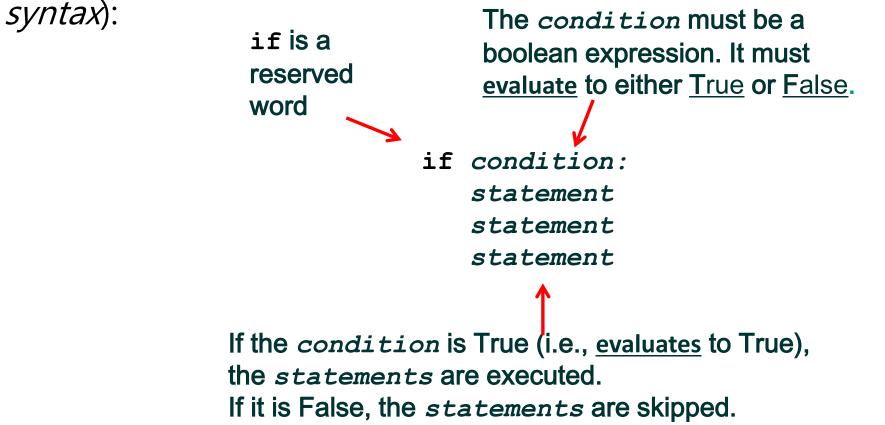
• There are a number of variations on boolean conditional structures, but these are the most important two:

if



The if Statement

• An *if statement* has the following form (*example below uses <u>Python</u>*



The if Statement

• An Python example of an **if** statement:

```
if sum > MAX:
    delta = sum - MAX
print ("The sum is " + str(sum))
```

- First the condition is evaluated -- either the value of sum is either greater than the value of MAX, or it is not
- If the condition is True, the assignment statement is executed -- if False, it is not
- The print statement, <u>not</u> being contingent upon sum < <u>MAX</u>, is always executed next

The if-else Statement

 An *else clause* can be added to an *if* statement to make an *if-else* statement

if condition:
 statement-block-1
else:
 statement-block-2
condition is True => statement-block-1
is executed
condition is False => statement-block-2
is executed

• One or the other will be executed, but not both

Repetition Statements

- Repetition statements better known as <u>loops</u> allow us to execute code multiple times
- The repetition (like branching) is controlled by <u>boolean</u> expressions that determine when it ends
- There are two basic kinds of loops:
 - o Indefinite (while)
 - Definite (for)
- The programmer should choose the right kind of loop for the situation

The while Loop

• A while *loop* has the following form (Python syntax):

while condition: statement statement

- If condition is True, statements are executed
- Then condition is evaluated again, and if it is still True, statement is executed again
- statements are executed repeatedly until condition becomes False

Indeterminate vs Determinate Loops

- A while loop will continue to run until its continuation condition becomes False.
- In theory, what stops the loop is a result of what happens during loop execution, so we may not yet know how many times the loop code should execute, so the while loop is <u>indeterminate</u>
- Other times, however, we will be able to determine this in advance – which means we can use a <u>determinate</u> loop

The for Loop

• A for *loop* has the following syntax:

The collection is The variable the series of objects refers to the current being processed item being processed for variable in collection: statement statement The statements are statement executed for the statement current item

The for Loop

• An example of a **for** loop:

for count in range(5):
 print (count)

- The **variable** section can be used to declare a variable for counting
- Like a while loop, the execution is dependent on a <u>condition</u> (here, implicit)
- Therefore, the body of a **for** loop will execute **<u>0+ times</u>**

The for Loop

• An example of a **for** loop:

for count in range(5):
 print (count)

- The **variable** section can be used to declare a variable for counting
- Like a while loop, the execution is dependent on a <u>condition</u> (here, implicit)
- Therefore, the body of a **for** loop will execute **<u>0+ times</u>**

Data Structures

- We have two basic structures for organizing many pieces of data:
 - **1. Numbered Sequences**: Here, you have a series of elements in a list (a.k.a., "tuple", "array", etc.), where you can query the sequence for a single element (or range) according to positional number.
 - 2. Data Maps: A collection of data pairs, where one half is the "key" and the other half is the "value". The <u>key</u> is used for looking up the <u>value</u>. (A.k.a., "dictionary", "hash", etc.)
- Using these more advanced structures, you can organize data in increasingly sophisticated ways within a program.

Introduction to Arrays - Java example

 We can declare a whole group (called an <u>array</u>) of variables of a specific type

```
int[] nums = new int [5];
```

```
char[] chars = new char[10];
```

• You can have arrays of <u>objects</u>, as well

String[] strings = new String[5];

 Note: Those variables in the arrays have not been initialized yet.

Introduction to Arrays - Java example

 To assign values to each variable, we can use a forloop:

```
for (int i = 0; i < 5; i++) {
  nums[i] = some valid integer expression;
}</pre>
```

 A single variable can be selected using an integer expression or value inside the []:
 count = 8;

```
int result = nums[count];
int otherResult = nums[count * 3 % 5];
```

Arrays and Initializer Lists

• An array can be defined and initialized with an an initializer list (an <u>array literal</u>):

char [] vowels = { `a', `e', `i', `o', `u' };

- Java allocates right amount of space based upon the list size
- An initializer list can be used only when the array is first declared, as above
- Because of Python's dynamic typing, this would be a non-issue:
 vals = ('a', 'e', 'i', 'o', 'u')
 vals = (1, 2, 3, 4, 5)
 vals = ("hello", "world", "goodbye")
 ...and so forth

Arrays and Loops

 Now we can coordinate the processing of one variable with the execution of one pass through a loop using an index variable, e.g:

```
int MAX = 5; // symbolic constant
```

```
int[] nums = new int[MAX];
```

```
for (int i = 0; i < MAX; i++) {
```

```
// use i as array index variable
```

Java statements using nums[i];

```
• Python equivalent: for i in nums:
```

```
# statements using nums[i]
```

Arrays and Loops

- Arrays are objects (only without a class)
- Each array has an *attribute* "length" that we can access to get the length of that array, e.g., nums.length == MAX: int MAX = 5; // symbolic constant

• Python equivalent: **len (nums)**

Dictionaries

- In addition to sequences, another useful way to organize data is in terms of <u>key-value pairings</u>
- This is the case with a <u>dictionary</u>, where data is organized like so:

key1	\rightarrow	value1
key2	\rightarrow	value2
key3	\rightarrow	value3
	•	

You can then use a specific <u>key</u> to retrieve a particular <u>value</u> from the dictionary.

Creating Dictionaries

$$key1 \rightarrow value1$$
$$key2 \rightarrow value2$$

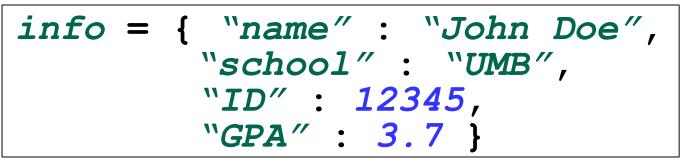
• <u>Syntax:</u>

variable =	first_key second_key	•	first_value, second_value,
	last_key	•	<pre>last_value }</pre>

- Keys must be of an <u>immutable</u> type, but values can be of <u>any</u> type
- Each key in the dictionary <u>must be unique</u>; otherwise, duplicated keys would create ambiguity

Using Dictionaries

• Let's create a dictionary:



• Now, we can...

Fetch a value by key:

print("My name is: " + info["name"])

My name is: John Doe

See if key exists:

print("Has major: " + str("major" in info))

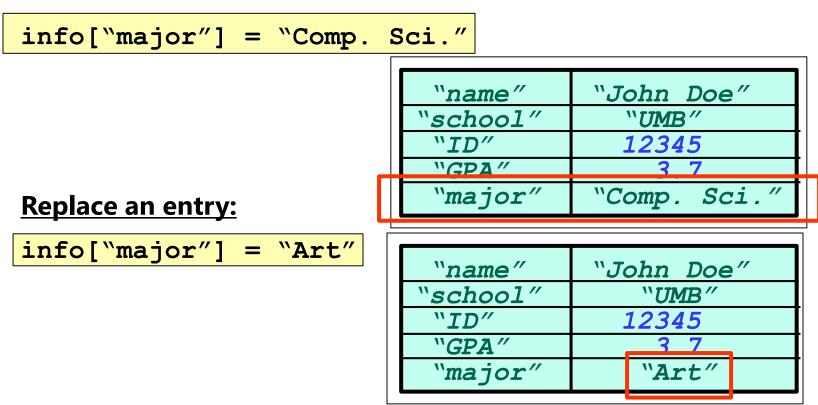
Has major: False

46

Using Dictionaries

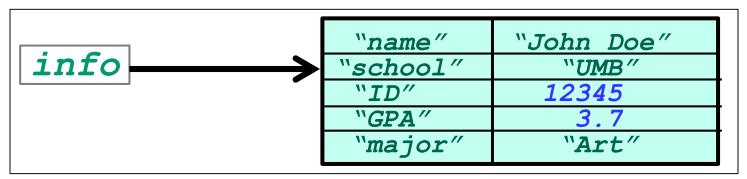
info	"name" "school" "ID"	"John Doe" "UMB" 12345
	``GPA″	3.7

Add a new entry (key-value pair):



47

Using Dictionaries



Delete an entry by key:

del info["major"]

Ο

6		
"name"	<i>"John Doe"</i>	
"school"	`` <i>UMB″</i>	
" <i>ID</i> "	12345	
``GPA″	3.7	

Fetch a value by key (with default):

0

print("Major:" info.get("major", "Undeclared")

0

Major: Undeclared