Computer Science Principles

Unit 3: Algorithms and Programming

LECTURE 4: PROGRAM STRUCTURE (U4)

DR. ERIC CHOU

IEEE SENIOR MEMBER

Topics

Variables

SECTION 1

Variables and Assignments

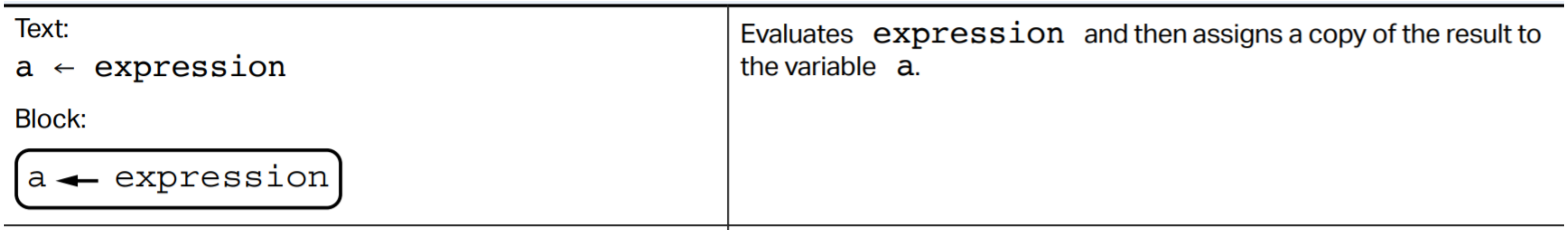
LECTURE 1

Variables

- A **variable** is a placeholder in your program for a value, just like in math. Variables are usually represented by letters or words.

Variables in Pseudocode and Python

- You can assign values to variables and also change these values through the assignment operator.
- The College Board Pseudocode uses the arrow ←.



Text: `a ← expression` Block: `a ← expression`	Evaluates `expression` and then assigns a copy of the result to the variable `a`.

Variables in Pseudocode and Python

- If you want to change the value the variable holds, you just have to reassign it.
- In Python, the = symbol is used to assign values to variables.

```
a = expression
```

- For all programming languages, the value that a variable has is the most recent one it's been assigned. Take this example (from the College Board's CED):

```
number = 7
changed_number = number
number = 17
print(changed_number)
```

Variables in Pseudocode and Python

- In this case, the print statement would return 7 because the variable **changed_number** was assigned to the value of the variable **number** when **number** still equaled 7.
- It can be difficult to keep track of what values your variables have, especially if you're changing them around a lot. Using extra print statements and hand tracing (from Big Idea 1) can help reduce this uncertainty.

Data Types

- **Data types** are different categories of data that your computer can represent. Examples of data types are **integers, strings, lists and booleans**.
- Numerical data is represented in several different ways. The numerical types in Python are known as int, float, (and complex.) You'll mainly be working with integers in AP CSP. **Integers** can only be positive or negative. You might come across **float**, or floating point numbers, as well: floating point numbers allow for decimal values to be represented.

```
int_example = 5 float_example = 5.52
```

String

- You can think of **strings** as lists of characters. In Python, strings are represented by quotation marks. You'll usually see them in the format of letters or words. However, strings can also be numerical; anything between quotation marks is considered a string.

```
"Hello world!"
"Fiveable"
"123456789"
"y + x"
```

Lists - Arrays

- A **list** is an ordered sequence of elements. They're also known as **arrays**. They allow you to treat multiple items as a single value, and can contain both words and numbers.

```
list_example = [Fish, fish, fish]
num_list_example = [2, 4, 6, 8]
```

Boolean

- Finally, the Boolean data type only represents two values: true or false.

```
Boolean_value = True
```

- The data type your variable is categorized under will determine what you can do to it. For example, you can't do math on strings, even if the string has a number in it.

Python

- Here's an example in Python:

```
y = 25
x = 5
print(y / x)
#prints a value of 5

y = "25"
x = "5"
print (y / x)
#returns an error/doesn't work
```

- Because of this, some values are better represented as one type of data than another.

Python

- Each variable can only hold one data value at a time, but that value itself could have multiple values within it. For example, you can store multiple values in a list, and then you can assign that list to the variable.
- When naming your variables, it's important to use meaningful, descriptive names. Variables with clear names are like good documentation (mentioned in Big Idea 1); they help make your code easy to understand.
- This isn't as vital in shorter programs, where you'll often see variables represented as just letters, but it's good practice for when you have to write longer programs with more lines to keep track of.

Python

- It's important to note that when naming variables, spelling and capitalization matter.

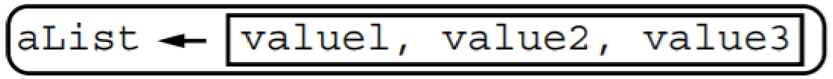
```
number = 7
#Is a different variable from

Number = 7
#and beware of typos!
```

Data Abstraction

LECTURE 2

Lists in APCSP Pseudocode

Text:
`aList ← [value1, value2, value3, ...]`
Block:

`aList ← value1, value2, value3`
	Creates a new list that contains the values `value1`, `value2`, `value3`, and `...` at indices `1`, `2`, `3`, and `...` respectively and assigns it to `aList`.

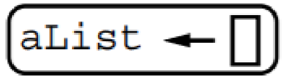
Lists in APCSP Pseudocode

- An **element** is an individual value in a list: in the example above, *value1, value2*, and *value3* are all elements. Each element is assigned an **index value**, or a number representing their place in the list.
- In the AP CSP Pseudocode, index numbers start at 1, so in the example above *value1* would have an index of 1, *value2* an index of 2, and so on. Index values are commonly used to reference and to work with the elements in a list.

```
This is a key difference from Python, where the
index values start at 0.
```

- You saw in the last example how you can assign a filled-in list to a variable.

Assign an Empty list to a Variable

Text: `aList ← [ ]` Block: 	Creates an empty list and assigns it to `aList`.

Assign One List to Another.

Text: `aList ← bList` Block: `aList ← bList`	Assigns a copy of the list `bList` to the list `aList`. For example, if `bList` contains `[20, 40, 60]`, then `aList` will also contain `[20, 40, 60]` after the assignment.

Data Abstraction

- Lists can be used to create data abstraction. Data abstraction simplifies a set of data by representing it in some general way. You can then work with that representation instead of each piece of data itself.
- For a very basic example, let's say you want to make a program that will print your to-do list out. You could manually insert values into your program to print, like so:

```
print ("My To-Do List is:" " Write essay", \
"Read Chapter 2", "Finish Math HW", \
"Attend Fiveable Class")
```

Data Abstraction

- But what if you made a list instead?

```
to_do = ["Write essay", "Read Chapter 2", \
"Finish Math HW", "Attend Fiveable Class"]
print ("My To Do List is:")
print (to_do)
```

- Now, instead of working with the data itself (the things on your to-do list), you're working with a variable, **to_do**, that represents the data.

Data Abstraction

- There are several advantages to this. For one, it makes your code neater and easier to read. It also allows you to edit the data easily. If you wanted to print a different list in the first example, you would have to erase your code entirely and rewrite it.
- By using a variable to represent the list, all you have to do is create a new list under a different name and use that variable instead.

Data Abstraction

```
new_to_do = ["Read Gatsby", "Practice Driving", "Go for walk"]

print ("My To Do List is:")
print (new_to_do)
```

- Using a list also allows you to work easily with individual elements within it, as you'll see below.
- By using data abstraction, you can create programs that are easier to build and maintain.

Mathematical Expressions

LECTURE 3

Algorithm

- An algorithm is a set of instructions used to accomplish a specific task or solve a problem.
- Sound familiar? The definition of an algorithm is very close to the definition of a program. That said, there are some major differences. The key difference is that an algorithm represents the problem-solving logic, while a program is how you carry it out. Programs execute algorithms.
- Going back to our cake metaphor from Big Idea 1, an algorithm would represent the steps you take to make the cake while a program would represent the written recipe that the baker, or computer, uses to make the cake.

Algorithm

- Algorithms can be expressed in a wide range of ways. They can be formally written out in a programming language like C++ or Java (or a block-based language like Scratch!) for a computer to execute, written out in Pseudocode or a natural language like English, or even drawn out in a diagram.
- Your computer will only do exactly what you tell it to, so it's important to make sure you're being clear and detailed when transferring your algorithms from your head to a program.

Sequencing

- Every algorithm is created using three basic concepts: **sequencing, selection,** and **iteration.** These concepts are conveyed in **code statements**.
- **Sequencing** consists of steps in your algorithm that are implemented in the order they're written in. Once you execute one statement, you go on to the next one.

Sequencing

```
first_number = 7
second_number = 5
sum_value = first_number + second_number
print (sum_value)
```

- Each of these statements are run in sequential order: the computer reads and processes them one after the other.
- We'll talk more about selection and iteration later in the guide.

Expressions

- Like you learned in math class, an **expression** is a statement that returns only one value.
- It can consist of a value, a variable, an operator or a procedure call that returns a value.
- Programs work with arithmetic operators: your good old-fashioned addition, subtraction, multiplication and division signs.
- Here are the operators in College Board's Pseudocode.

Text and Block:
`a + b`
`a - b`
`a * b`
`a / b`
	The arithmetic operators `+`, `-`, `*`, and `/` are used to perform arithmetic on `a` and `b`.
	
	For example, `17 / 5` evaluates to `3.4`.
	
	The order of operations used in mathematics applies when evaluating expressions.

Text and Block: `a MOD b`	Evaluates to the remainder when `a` is divided by `b`. Assume that `a` is an integer greater than or equal to `0` and `b` is an integer greater than `0`. For example, `17 MOD 5` evaluates to `2`. The `MOD` operator has the same precedence as the `*` and `/` operators.

Expressions

- You'll notice the addition of a new operator known as the **MOD** operator. This is short for the Modulo operator. In the a MOD b format, a is divided by b and MOD gives you what the remainder would be. For example, 13 MOD 3 would return 1 because 13 divided by 3 can be written as 4 remainder 1.

🔗 **Code.org** has created a visual to represent how MOD works known as the **Modulo Clock.**

- Evaluating an expression is done through an order of operations specified by the programming language you're working with. It's generally going to be PEMDAS, like it is in mathematics. In College Board's Pseudocode, precedence as the multiplication and division operators. (PEMMDAS?) the MOD operator has the same

❗Don't forget that, in the PEMDAS system, it's multiplication **or** division and addition **or** subtraction, depending on what order they're written in the equation. Liberal use of parentheses can help make some of this less confusing.

Strings

LECTURE 4

What are Strings?

Strings, as mentioned earlier, are an ordered list of characters. You can navigate through them using index values, just like how you can navigate through a list using index values.

Although you can't perform mathematical operations with strings, you can still manipulate them.

A **substring** is a part of an existing string. For example, "APcomputer" would be a substring of the string "APcomputerscienceprinciples." The process of creating a substring is known as **slicing**.

Example

- There are many different ways to slice a string: here's a basic example.

```
string_example = "APcomputerscienceprinciples"
print (string_example[0:10])
```

- This code returns the substring "APcomputer." In Python, index values begin at 0, and the character at the end index number (10) is not included.
- You can also concatenate strings. **String concatenation** occurs when two or more strings are joined end to end to make a new string. This is usually represented by the + symbol.

Concatenation Example

```
part_one = "Hello"
part_two = "_World!"
print (part_one + part_two)
```

The code returns:

```
Hello_World!
```

Boolean Expressions

LECTURE 5

Boolean Variables

- **Boolean variables** can only represent the values true or false. **Relational operators** are used with Boolean values to test the relationship between two values.

Relational and Boolean Operators	

Text and Block:
`a = b`
`a ≠ b`
`a > b`
`a < b`
`a ≥ b`
`a ≤ b`
	The relational operators =, ≠, >, <, ≥, and ≤ are used to test the relationship between two variables, expressions, or values. A comparison using relational operators evaluates to a Boolean value.
	
	For example, `a = b` evaluates to `true` if `a` and `b` are equal; otherwise it evaluates to `false`.

Python

- When you compare two values using a relational operator, the computer will return a Boolean value.

- Take this Python example below:

```
y = 5
x = 55
print (x > y)
```

- The computer will print the word True because 55 is greater than 5.

Not, And, Or

- Boolean values are also used with **logical operators** known as NOT, AND and OR. These can be used to combine multiple conditions, whereas relational operators can only evaluate one condition.
- The NOT operator is used to reverse what the condition evaluates to. If a condition is true, the operator will evaluate to false, and vice versa.

```
y = 5
x = 55
print (not y >= 5)
```

- In this case, the computer will print the word "False." y is equal to 5. This makes the condition true—it's the not operator that reverses this so the computer returns false.

Not

- The NOT operator is used to reverse what the condition evaluates to. If a condition is true, the operator will evaluate to false, and vice versa.

```
y = 5
x = 55
print (not y >= 5)
```

- In this case, the computer will print the word "False." y is equal to 5. This makes the condition true—it's the not operator that reverses this so the computer returns false.

NOT operator in Pseudocode

Text: `NOT condition` Block: NOT $\boxed{\texttt{condition}}$	Evaluates to `true` if `condition` is `false`; otherwise evaluates to `false`.

AND

- The AND operator is used to combine two conditions. The operator will only evaluate to true if both conditions are met.

```
y = 5
x = 55
print (y >= 5 and x <= 44)
```

- The computer will print the word "False" because x is not less than or equal to 44, even though y equals 5.

AND operator in Pseudocode

Text:
`condition1 AND condition2`
Block:
`condition1` AND `condition2`
	Evaluates to `true` if both `condition1` and `condition2` are `true`; otherwise evaluates to `false`.

OR

- The OR operator also involves two conditions. In this case, the operator will evaluate to true if one condition or the other is met.

```
y = 5
x = 55
print (y >= 5 or x <= 44)
```

- The computer will print the word "True" because, although the second condition isn't met, the first one is.

OR operator in Pseudocode

Text:
`condition1 OR condition2`
Block:
`(condition1)  OR  (condition2)`
	Evaluates to `true` if `condition1` is `true` or if `condition2` is `true` or if both `condition1` and `condition2` are `true`; otherwise evaluates to `false`.

Variables Explore

LESSON 1 [CODE.ORG]

Variables Investigate

LESSON 2 [CODE.ORG]

Variables Practice

LESSON 3 [CODE.ORG]

Variables Make

LESSON 4 [CODE.ORG]

Conditionals

SECTION 2

Conditionals

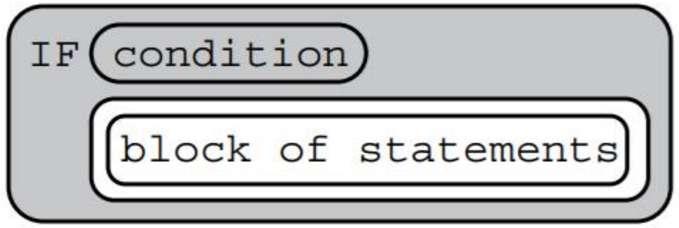
LECTURE 6

Conditionals

- Earlier, we introduced the concept of sequencing, selection and iteration. The **selection** process primarily takes the form of conditional statements known as **if statements**. Certain requirements, or **conditions**, need to be met in order for selection statements to run.
- If statements execute different segments of code based on the result of a Boolean expression. Here's the format of a basic if statement in Pseudocode...

Pseudocode Example

Text:	The code in `block of statements` is executed if the Boolean expression `condition` evaluates to `true`; no action is taken if `condition` evaluates to `false`.

```
IF(condition)
{
 <block of statements>
}
```
 Block: 	

Python

```
strawberries_in_fridge = 7

if strawberries_in_fridge >= 7:
  print ("You can make strawberry shortcake!")
```

- The code in the if statement is executed only if the condition is met. In this case, the condition is the boolean expression strawberries_in_fridge ≥ 7. The condition is true, so the code inside the if statement (the print statement) will execute.
- If strawberries_in_fridge was less than seven in this case, the program would just skip over the code inside the statement and continue to the next part of the program.

Else

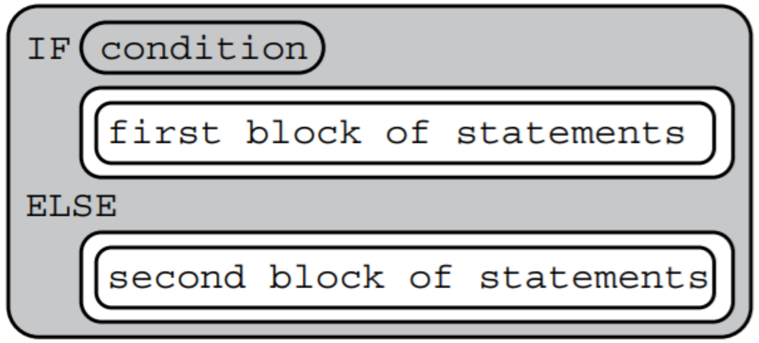
- However, if statements can also have else statements attached to them that specify what would happen if the condition wasn't met. Here they are in Pseudocode...

Else

Text:

```
IF(condition)
{
 <first block of statements>
}
ELSE
{
 <second block of statements>
}
```

Block:



The code in `first block of statements` is executed if the Boolean expression `condition` evaluates to `true`; otherwise the code in `second block of statements` is executed.

Python

```
strawberries_in_fridge = 7

if strawberries_in_fridge >= 10:
   print ("You can make strawberry shortcake!")
else:
   print ("Sorry, no shortcake for you!")
```

- Else statements will only run when the condition attached to an if statement comes back false. In this case, because the if statement isn't true, the program will run the else statement instead.

Nested Conditionals

LECTURE 7

Nested Conditional Example

```
strawberries_in_fridge = 7
number_of_eggs = 12

if strawberries_in_fridge >= 7:
  print ("You can make strawberry shortcake!")
   if number_of_eggs =/ 12:
          print ("... if you go to the store first.")
   else:
•         print ("So start baking!")
```

- In this example, the program returns:

```
You can make strawberry shortcake! So start baking!
```

Nested Conditional Example

- In this example, your program would return the line "You can make strawberry shortcake!" regardless of how many eggs you have because you have enough strawberries. It's only after this is confirmed that your program looks at the nested if statement.
- It's important to note that nested if statements are part of the larger if statement themselves: none of the code would run if **strawberries_in_fridge** was less than 7. (This also means you need to be careful about your spacing when you write nested if statements to ensure that everything nests correctly.)

Conditionals Explore

LESSON 5 [CODE.ORG]

Conditionals Investigate

LESSON 6[CODE.ORG]

Conditionals Practice

LESSON 7 [CODE.ORG]

Conditionals Make

LESSON 8 [CODE.ORG]

Function

SECTION 3

Calling Procedures

LECTURE 8

Parts of a Procedure

- Let's go back to this example we used earlier for a program that sums two numbers.

```
first_number = 5
second_number = 7
sum_value = first_number + second_number
print (sum_value)
```

- It goes without saying that, if you wanted to sum more than one set of numbers at a time, you'd have to rewrite these lines over and over again...

Parts of a Procedure

- Let's see how we'd go about writing this as a function, or a procedure in Python.
- A function in Python looks like this:

```
def name_of_function (parameter1, parameter2):
  #code statements here
```

- So, our summing machine would look like this!

```
def summing_machine(first_number, second_number):
  print (first_number + second_number)
```

Parts of a Procedure

- What are those parentheses for? Procedures often require some sort of parameter in order to work. Parameters are the input variables of a procedure. In this case, the parameters are the two numbers you want to add together: first_number and second_number.

  ! You don't always need to use parameters in a procedure.

- When you call a procedure, your program acts like those lines of code are written out. Once it's done executing those code lines, your program goes back to reading code sequentially.

Parts of a Procedure

- When you call that procedure with defined values or variables, those values are known as arguments.

```
def summing_machine(first_number, second_number):
  print (first_number + second_number)


summing_machine(5, 7)
#In this example, 5 and 7 are examples of arguments.
```

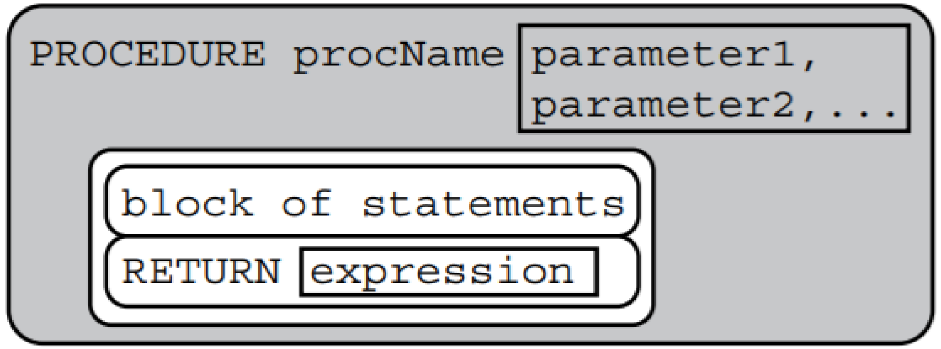
- If you have a procedure in a longer chunk of code and you want to exit the function, you can use the return statement, in both Python and Pseudocode. This will return the value of an expression that you can then use, without needing to print it.

Parts of a Procedure

Text:

```
PROCEDURE procName(parameter1,
                   parameter2, ...)
{
 <block of statements>
 RETURN(expression)
}
```

Block:



Defines `procName` as a procedure that takes zero or more arguments. The procedure contains `block of statements` and returns the value of `expression`. The `RETURN` statement may appear at any point inside the procedure and causes an immediate return from the procedure back to the calling statement.

The value returned by the procedure `procName` can be assigned to the variable `result` using the following notation:

```
result ← procName(arg1, arg2, ...)
```

Parts of a Procedure

- For example:

```
def summing_machine(first_number, second_number):
    value = first_number + second_number
    return (value)
answer = summing_machine(5, 7)
print (answer)
#you can use this to manipulate the value the function returns as well.
print (answer + 1)
```

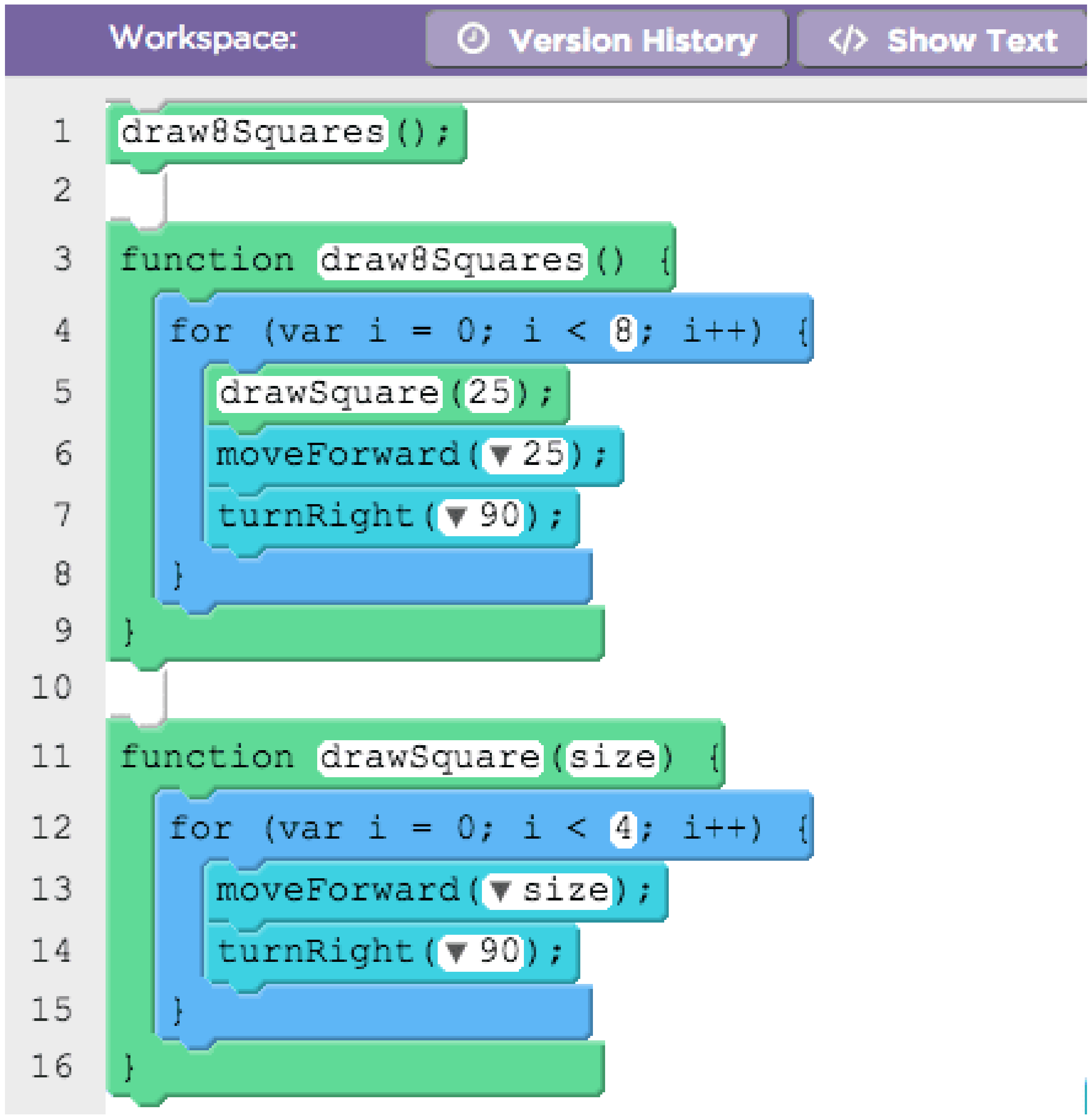
- Note that the Pseudocode allows you to assign this returned value to a variable using an arrow.

Developing Procedures

LECTURE 9

Developing Procedures - Modularity

- Procedures are an example of abstraction. (In fact, it's part a special form of abstraction known as **procedural abstraction**.) This is because you can call a procedure without knowing how it works. Indeed, you often do: a common built-in procedure in programming languages is the print() procedure.
- Procedures allow you to solve a large problem based on the solution to smaller sub-problems. For example, my final Create project was a choose-your-own-adventure story. In order to make this story, I had to figure out how to let players choose what path they were going to take, how to display this path choice, how to track a player's progress in the game so their actions would have consequences... and so on. I solved the larger "problem" of writing a choose-your-own-adventure story by solving all these smaller problems with procedures I then combined.
- When you divide a computer program into separate sub-programs, it's known as **modularity**.



```
draw8Squares();

function draw8Squares() {
  for (var i = 0; i < 8; i++) {
    drawSquare(25);
    moveForward(▼25);
    turnRight(▼90);
  }
}

function drawSquare(size) {
  for (var i = 0; i < 4; i++) {
    moveForward(▼size);
    turnRight(▼90);
  }
}
```

Readability

- Procedures can also help you simplify your code and improve its readability.
- Instead of this:

```
first_number = 7
second_number = 5
sum_value = first_number + second_number
print (sum_value)
first_number = 8
second_number = 2
sum_value = first_number + second_number
print (sum_value)
first_number = 9
second_number = 3
sum_value = first_number + second_number
print (sum_value)
```

Readability

- you get this:

```
def summing_machine(first_number, second_number):
  sum_value = first_number + second_number
  print (sum_value)

summing_machine(5, 7)
summing_machine(8, 2)
summing_machine(9, 3)
```

Readability

- Isn't the second example a lot easier to understand? This is especially important in larger programs, where you might already be looking at hundreds of lines of code.
- The beauty of procedures is how they can be reused. Parameters usually represent general categories of data, such as **numbers** or **strings**. This means you could use one procedure for a wide range of individual scenarios.

Readability

- **Procedural abstraction** also allows programmers the flexibility to modify or fix procedures without affecting the whole program, as long as the procedure continues to function in the same way. Furthermore, they can make a change or edit once, in the procedure, and it will be implemented everywhere where the procedure is called.

Functions Explore/Investigate

LESSON 9 [CODE.ORG]

Functions Practice

LESSON 10 [CODE.ORG]

Functions Make

LESSON 11 [CODE.ORG]

Project

SECTION 4

Decision Maker App

LESSON 12/13/14 [CODE.ORG]