Grade 6 Math Circles

Computer Science

Computer Science

Computer Science is the study of computers and algorithmic processes, including their principles, their hardware and software designs, their applications, and their impact on society. In these lessons, we will focus more on **computer programming**, which is the process of designing and building an executable computer program to accomplish a specific computing result or to perform a specific task. These programs are written in **programming languages**, which are similar to other languages in the sense that they are defined through the use of **syntax** and **semantic** rules, to determine structure and meaning, respectively.

Syntax is the grammatical structure of a language, whereas **semantics** is the meaning being conveyed. A sentence that is syntactically correct is not always semantically correct.

Example 1

The sentence "*That was a greatest film*" is syntactically incorrect because it uses incorrect grammar.

The sentence "*The snake picked it up with its hand*" is syntactically correct, but semantically incorrect because snakes do not have hands, meaning the sentence doesn't make sense.

Example 1 shows us examples of **syntax errors** and **semantic errors** in the English language. Similarly, programming languages also have syntax and semantic errors, which will be discussed later in the lesson.

There are a number of programming languages that are very popular in the world. Some of the top ones are **C**, **Java**, **Python**, **C++**, **C#**, **JavaScript**, **R** and **SQL**. The differences between programming languages is syntax, and depending on the languages, the differences can be either minimal or significant. For the next two lessons, we will focus on **Python**.

Python

Python is a high-level general-purpose programming language, with a design philosophy that emphasizes code readability with its use of significant indentation. Because of this, Python is a very useful

programming language for small-scale and large-scale projects. In addition, there are no restrictions to what we are allowed to create a program for, thus, Python has an infinite number of uses. We create these programs in Python through **coding**.

Coding is the process of creating instructions for computers using programming languages. To put it simply, it is our way of telling the computer what we want it to do. And as stated above, this can be as small as one line of code, or as large as millions of lines of code. Coding can prove to be very frustrating at times because computers are both very smart and very dumb, in the sense that they can perform many computations almost instantaneously, but they require exact instructions in order to be able to execute them properly.

Let's start with learning how to define variables and assign values to them. Unlike elsewhere in mathematics, variables don't really mean the same thing in computer science. They are able to have any value, but they can only have one value at a time. To define a variable in Python is to just assign it a value. For example, if we want x to have a value of 6 in Python, then we would simply write the following line of code:

```
x = 6
```

An important thing to note here is that '=' does not necessarily mean equality in Python. Instead, '=' is used to assign values. Specifically, it assigns the value of the right-hand side as the value of the left-hand side. So, if we have the variables x and y and we write the code $x = y$, this means that the value of x is set to the value of y. It does not mean that x and y are the same. If we later change the value of y, then the value of x does not also change, it remains the same.

Example 2

What are the values of a, b and c after the following code is run?

```
a = 3
b = 8
c = -4
a = c
```

Solution 2

We will keep track of each value after each step using the table below:

	Initial	Line 1	Line 2	Line 3	Line 4
a	None	3	3	3	-4
b	None	None	8	8	8
c	None	None	None	-4	-4

Thus, after the code is run, we have that $a = -4$, $b = 8$ and $c = -4$.

Note that *None* is not a value for a variable, it simply represents that the variable currently has no value. Also, after Line 4 is executed, the value of a changes from 3 to -4. This is because when the line $a = c$ is executed, the previous value of a is overwritten and now set as the value of c.

This processing of keeping track of the values of variables while the code is executed is known as **tracing**. It can be very helpful in cases where programs are not producing the desired outcomes because we can follow the steps to see where it may be going wrong. Tracing can be done by hand, like in Example 2, or it can be done by using a **stepper**, which is a program that traces the code for you and walks you through the execution. One such stepper is **Python Tutor**, which is linked here. Unless you prefer to download Python-compatible software, Python Tutor will be the main resource you use over the next two lessons for coding.

A short tutorial video for Python Tutor is linked here, which will go over Example 2.

Activity 1

What are the values of a, b, c and d after the following code is run?

```
a = 0
b = a
d = b
a = 7
c = a
```

Data Types

Data types are the classification or categorization of data items in computer programming. Same (or similar) data types can be compared, while different data types cannot be compared. There are many data types in Python, but for these lessons we will only be focusing on the following four types:

- **int** $\Rightarrow$ integers, all positive and negative integers (i.e., -1)
- **float** $\Rightarrow$ all positive and negative numbers represented as decimals (i.e., 2.0)
- **str** $\Rightarrow$ strings, all text enclosed in quotations (i.e., 'hello' or "hello")
- **bool** $\Rightarrow$ boolean, *True* or *False*

Activity 2

Determine the data type of the following values in Python.

(a) -4839.1

(b) "76"

(c) 0

(d) 'False'

We can change the data types of variables with the following:

Code	Description	Example
$int(a)$	If a is **float**, then decimal is dropped If a is **str**, then quotations are dropped	$int(3.7) \Longrightarrow 3$ $int(\text{"1"}) \Longrightarrow 1$
$float(a)$	If a is **int**, then .0 is added If a is **str**, then quotations are dropped	$float(2) \Longrightarrow 2.0$ $float(\text{"9.3"}) \Longrightarrow 9.3$
$str(a)$	If a is **int** or **float**, then quotations are added	$str(17) \Longrightarrow$ "17" $str(7.4) \Longrightarrow$ "7.4"

Python Operators

In order for computer programs to be able to perform computations, we must have a way for values to be compared or interact with one another. We are able to do this with **Python operators**. Below are three tables that define different operators used in Python, along with examples for them. These

tables are separated based on the types of operators.

For the following tables, let $a = 5$, $b = 2$, $c = True$ and $d = False$.

Arithmetic Operators

Code	Description	Example
+	Adds values on either side of the operator	$a + b \Longrightarrow 7$
−	Subtracts right-hand operand from left-hand operand	$a - b \Longrightarrow 3$
*	Multiplies values on either side of the operator	$a * b \Longrightarrow 10$
/	Divides left-hand operand by right-hand operand	$a / b \Longrightarrow 2.5$
//	Divides left-hand operand by right-hand operand and returns the integer quotient	$a // b \Longrightarrow 2$
%	Divides left-hand operand by right-hand operand and returns the remainder	$a \% b \Longrightarrow 1$
**	Left-hand operand to the power of right-hand operand	$a ** b \Longrightarrow 25$

Note that these operators can only be applied on values that are **int** or **float**. The one exception is + which can be applied on two **str** values (e.g. "*good*" + "*night*" $\Longrightarrow$ "*goodnight*").

Comparison Operators

Code	Description	Example
==	If the values of two operands are equal, then the condition becomes True	$a == b \Longrightarrow False$
!=	If the values of two operands are not equal, then the condition becomes True	$a != b \Longrightarrow True$
>	If the value of the left operand is greater than the value of the right operand, then the condition becomes True	$a > b \Longrightarrow True$
<	If the value of the left operand is less than the value of the right operand, then the condition becomes True	$a < b \Longrightarrow False$
>=	If the value of the left operand is greater than or equal to the value of the right operand, then the condition becomes True	$a >= b \Longrightarrow True$
<=	If the value of the left operand is less than or equal to the value of the right operand, then the condition becomes True	$a <= b \Longrightarrow False$

Boolean/Logical Operators

Code	Description	Example
and	If both operands are true, then the condition becomes True	$(c \text{ and } d) \Longrightarrow False$
or	If any of the two operands are true, then the condition becomes True	$(c \text{ or } d) \Longrightarrow True$
not	Used to reverse the logical state of its operand	$\text{not}(c \text{ and } d) \Longrightarrow True$ $\text{not}(c \text{ or } d) \Longrightarrow False$

These three operators can be thought of in terms of probability concepts: **and** is similar to intersections, **or** is similar to unions, and **not** is similar to complements.

Take some time to try out all these operators in Python Tutor (or another resource) so that you are comfortable using them. Note, that when you perform an arithmetic operator with an **int** and **float**, the result will be a **float**.

Example 3

Let $a = 9$, $b = True$, $c =$ "$math$", $d = False$, $e = 4.0$ and $f =$ "$circles$". Determine the following:

(a) $f + c$

(b) $\text{not}(b)$

(c) $(a * e) - (a \;//\; e)$

(d) $(a > e)$ or $(d \text{ and } b)$

Solution 3

Python Tutor can solve these instantaneously, so let's do it by hand to see how we get each result.

(a) $f + c \Longrightarrow$ "$circles$" + "$math$" $\Longrightarrow$ "$circlesmath$"

(b) $\text{not}(b) \Longrightarrow \text{not}(True) \Longrightarrow False$

(c) $(a * e) - (a \;//\; e) \Longrightarrow (9 * 4.0) - (9 \;//\; 4.0) \Longrightarrow 36.0 - 2.0 \Longrightarrow 34.0$

(d) $(a > e)$ or $(d \text{ and } b) \Longrightarrow (9 > 4.0)$ or $(False \text{ and } True) \Longrightarrow True$ or $False \Longrightarrow True$

Programs

So far, we've covered defintions, data types, how to define variables, tracing, and how to perform operations. But, we haven't discussed creating programs to compute specific results or perform specific tasks, which is the essence of computer programming.

For example, how could we write a program that gives us the sum of the digits of a 4-digit number?

Below is a general outline for a computer program in Python.

```
def program_name(parameter_1, parameter_2, ..., parameter_n):

    # here is where we write the body of the program, including
    # defining variables, operations and computations

    # once we've reached our desired outcome, we can either
    # return it, print it, or neither (depending on program)

    return desired_outcome
    # or
    print(desired_outcome)
    # or
```

The first line of each program begins with **def**, followed by the name of the program (program_name). Next to the program name, inside the brackets, are the variables that we want to input into the program, called **parameters**. At the end of the first line we write ":"

Next, comes the **body** of the program, which is indented. Here is where we write all the necessary code in order for our program to compute what we want it to compute.

After we have our desired outcome, we can either return it using **return**, or print it using **print()**. Returning outcomes is more useful when we wish to use the outcome for additional computations, and printing is more commonly used when we just want to know the outcome.

The green lines of code above are called **comments**. We can write comments by writing "#" and then everything else we write after on that line will be a comment. Python will ignore comments when a program is executed. They are simply there for us, making the code easier to read and understand.

Example 4

Write a program called *sum_digits* that inputs a positive 4-digit number and outputs the sum of the digits. What is returned when we run the following code?

(a) *sum_digits*(4391)

(b) *sum_digits*(1001)

(c) *sum_digits*(9999)

(d) *sum_digits*(2021)

Solution 4

```
def sum_digits(num):          # defining program sum_digits

    ones = num % 10                 # digit in the ones place
    tens = (num // 10) % 10         # digit in the tens place
    hundreds = (num // 100) % 10    # digit in the hundreds place
    thousands = num // 1000         # digit in the thousands place

    sum = ones + tens + hundreds + thousands  # sum of the digits

    return sum   # returns the sum of the digits
```

(a) *sum_digits*(4391) returns 17

(b) *sum_digits*(1001) returns 2

(c) *sum_digits*(9999) returns 36

(d) *sum_digits*(2021) returns 5

Here is a short video for solving Example 4 using Python Tutor.

Activity 3

In order to convert temperature from Celsius to Fahrenheit, first multiply the temperature in Celsius by 1.8 and then add 32. Write a Python program called *celsius_to_fahrenheit* that inputs a temperature in Celsius and outputs the corresponding temperature in Fahrenheit. What is the corresponding Fahrenheit temperature for the following?

(a) 0 °C

(b) 100 °C

(c) 18 °C

(d) −40 °C

Syntax and Semantics Errors

Within computer programming, **syntax errors** are errors that "break" the code. This means that if you try to run a computer program that has syntax errors, the code will not be able to execute. Syntax errors are easy to find because you will receive an error message that the code is unable to run. For example, trying to run the line of code:

$$a = 100 + \text{“}hello\text{”}$$

is a syntax error because we are attempting to add an **int** and a **str**.

Within computer programming, **semantic errors** are errors that arise from our own incorrect logic. This means that all our code is syntactically correct, but our method of solving the problem is incorrect, so our outcome will be incorrect. These can be very difficult to find because we will receive no error messages. The best way to find semantic errors is either by tracing or using a stepper, like Python Tutor. For example, suppose we have the following code for Example 4:

```
def sum_digits(num):

    ones = num % 10
    tens = num % 10
    hundreds = num % 100
    thousands = num % 1000

    sum = ones + tens + hundreds + thousands

    return sum
```

Here the code is all syntactically correct, so we won't get any error messages, but our logic for solving the problem is incorrect, so we have a semantic error. If we run the code

$$sum\_digits(4391)$$

instead of getting 17, as we should, we will get 484, which is clearly incorrect. In order to correct this mistake, we must trace the program to see where we went wrong, and then fix the error.

A final note for this lesson is the **input()** function, which incorporates user input while the code is being run. Observe the example below.

```
a = input("Enter a word:")
b = input("Enter a word:")

print(a + " " + b)
```

Here, when we run the code, we are presented with a textbox with the prompt: "Enter a word:". Whatever value is entered will be the value of a, which will be a **str**. The same will occur for b immediately after. After this, the code will run as normal and will print the value of a followed by the value of b, with a space in between. Try running this code on your own, as well as other cases using **input()**.