1.1 Programming (general)

Computer program basics

Computer programs are abundant in many people's lives today, carrying out applications on smartphones, tablets, and laptops, powering businesses like Amazon and Netflix, helping cars drive and planes fly, and much more.

A computer ***program*** consists of instructions executing one at a time. Basic instruction types are:

- ***Input***: A program gets data, perhaps from a file, keyboard, touchscreen, network, etc.
- ***Process***: A program performs computations on that data, such as adding two values like x + y.
- ***Output***: A program puts that data somewhere, such as to a file, screen, or network.

Programs use ***variables*** to refer to data, like x, y, and z below. The name is due to a variable's value "varying" as a program assigns a variable like x with new values.

PARTICIPATION ACTIVITY | 1.1.1: A basic computer program.

Animation content:

```
The animation executes the following computer program:

x = Get next input
y = Get next input

z = x + y
Put z to output

Input (keyboard) is as follows:
2 5

Output (screen) is as follows:
7
```


Animation captions:

1. A basic computer program's instructions receive input, process the input, and produce output. This program first assigns x with what is typed on the keyboard input, in this case, 2.

2. The program's next instruction assigns y with the next input, in this case 5.
3. The program then processes the input, in this case the program assigns z with x + y (so 2 + 5 yields z of 7).
4. Finally, the program puts z (7) to output, in this case to a screen.

PARTICIPATION ACTIVITY | 1.1.2: A basic computer program.

Consider the example above.

1) How many instructions does the program have?

Check Show answer

2) Suppose a new instruction were inserted as follows:

...
z = x + y
Add 1 to z (new instruction)
Put z to output

What would the last instruction output to the screen?

Check Show answer

3) Consider the instruction: z = x + y. If x is 10 and y is 20, then z is assigned with _____.

Check Show answer

A program is like a recipe

Some people think of a program as being like a cooking recipe. A recipe consists of *instructions* that a chef executes, like adding eggs or stirring ingredients.

Baking chocolate chip cookies from a recipe

- Mix 1 stick of butter and 1 cup of sugar.
- Add egg and mix until combined.
- Stir in flour and chocolate.
- Bake at 350°F for 8 minutes.

Likewise, a computer program consists of instructions that a computer executes, like multiplying numbers or outputting a number to a screen.

A first programming activity

Below is a simple tool that allows a user to rearrange some pre-written instructions (in no particular programming language). The tool illustrates how a computer executes each instruction one at a time, assigning variable m with new values throughout, and outputting ("printing") values to the screen.

PARTICIPATION ACTIVITY 1.1.3: A first programming activity.

Execute the program and observe the output. Click and drag the instructions to change the order of the instructions, and execute the program again. Not required, but can you make the program output a value greater than 500? How about greater than 1000?

Run program

```
m = 5

put m

m = m * 2
put m

m = m * m
put m

m = m + 15
put m
```

m:

PARTICIPATION ACTIVITY | 1.1.4: Instructions.

1) Which instruction completes the program to compute a triangle's area?
base = Get next input
height = Get next input
Assign x with base * height
____
Put x to output

- Multiply x by 2
- Add 2 to x
- Multiply x by 1/2

2) Which instruction completes the program to compute the average of three numbers?
x = Get next input
y = Get next input
z = Get next input
____
Put a to output

- a = (x + y + z) / 3
- a = (x + y + z) / 2
- a = x + y + z

Computational thinking

Mathematical thinking became increasingly important throughout the industrial age, enabling people to successfully live and work. In the information age, many people believe ***computational thinking***, or creating a sequence of instructions to solve a problem, will become increasingly important for work and everyday life. A sequence of instructions that solves a problem is called an ***algorithm***.

PARTICIPATION ACTIVITY | 1.1.5: Computational thinking: Creating algorithms to draw shapes using turtle graphics.

A common way to become familiar with algorithms is called turtle graphics: You instruct a robotic turtle to walk a certain path, via instructions like "Turn left", "Walk forward 10 steps", or "Pen down" (to draw a line while walking).

The 6-instruction algorithm shown below ("Pen down", "Forward 100", etc.) draws a triangle.

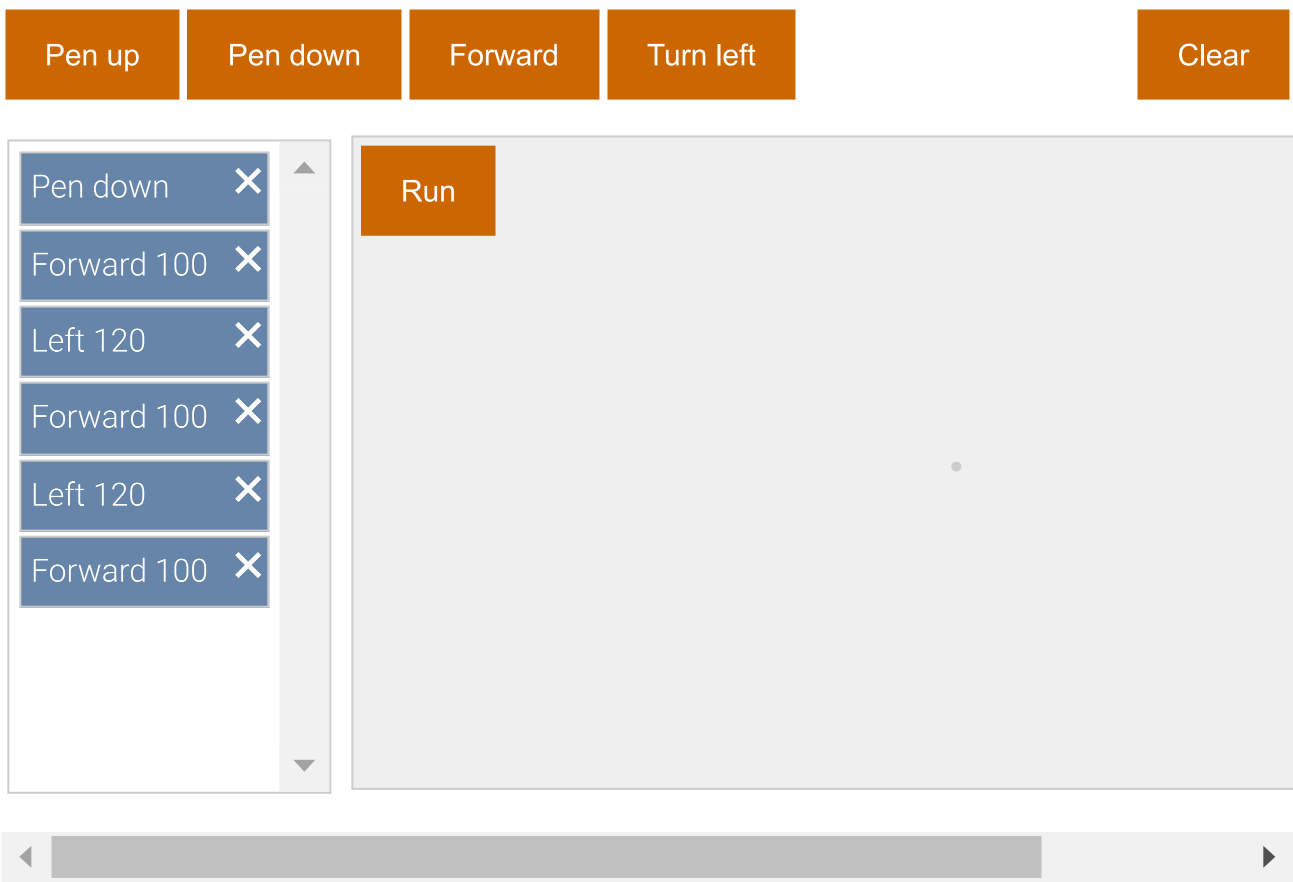
1. Press "Run" to see the instructions execute from top to bottom, yielding a triangle.

2. Can you modify the instructions to draw a square? Hint: "Pen down", "Forward 100", "Left 90", "Forward 100", "Left 90" -- keep going!
3. Experiment to see what else you can draw.

Note: The values after a Left or Right turn are angles in degrees.

How to:

- Add an instruction: Click an orange button ("Pen up", "Pen down", "Forward", "Turn left").
- Delete an instruction: Click its "x".
- Move an instruction: Drag it up or down.



1.2 Programming using Python

Python interpreter

The ***Python interpreter*** is a computer program that executes code written in the Python programming language. An ***interactive interpreter*** is a program that allows the user to execute one line of code at a time.

Code is a common word for the textual representation of a program (and hence programming is also called *coding*). A ***line*** is a row of text.

The interactive interpreter displays a ***prompt*** (">>>") that indicates the interpreter is ready to accept code. The user types a line of Python code and presses the enter key to instruct the interpreter to execute the code. Initially you may think of the interactive interpreter as a powerful calculator. The example program below calculates a salary based on a given hourly wage, the number of hours worked per week, and the number of weeks per year. The specifics of the code are described elsewhere in the chapter.

PARTICIPATION ACTIVITY 1.2.1: The Python interpreter.

Animation captions:

1. After each press of the enter key, the python interpreter executes the line of code.
2. The python interpreter can be used as a calculator and can perform a variety of calculations.
3. Users can change values and execute calculations again.

PARTICIPATION ACTIVITY 1.2.2: Match the Python terms with their definitions.

Line Code Prompt Interpreter

	A program that executes computer code.
	The text that represents a computer program.
	Informs the programmer that the interpreter is ready to accept commands.
	A row of text.

Reset

Executing a Python program

The Python interactive interpreter is useful for simple operations or programs consisting of only a few lines. However, entering code line-by-line into the interpreter quickly becomes unwieldy for any program spanning more than a few lines.

Instead, a programmer can write Python code in a file, and then provide that file to the interpreter. The interpreter begins by executing the first line of code at the top of the file, and continues until the end is reached.

- A ***statement*** is a program instruction. A program mostly consists of a series of statements, and each statement usually appears on its own line.
- ***Expressions*** are code that return a value when evaluated; for example, the code `wage * hours * weeks` is an expression that computes a number. The symbol * is used for multiplication. The names wage, hours, weeks, and salary are ***variables***, which are named references to values stored by the interpreter.
- A new variable is created by performing an ***assignment*** using the = symbol, such as `salary = wage * hours * weeks`, which creates a new variable called salary.
- The ***print()*** function displays variables or expression values.
- '#' characters denote ***comments***, which are optional but can be used to explain portions of code to a human reader.
- Many code editors color certain words, as in the below program, to assist a human reader in understanding various words' roles.

PARTICIPATION ACTIVITY | 1.2.3: Executing a simple Python program.

Animation content:

undefined

Animation captions:

1. The python interpreter reads a file line by line. Variables wage, hours, and weeks are named references that refer to values stored by the interpreter.
2. 20 * 40 * 52 is computed, and then the variable salary is assigned with the value.
3. The print statement prints 'Salary is:' to the screen and displays the value of the variable salary.
4. Values can be overwritten if the same variable name is used.

PARTICIPATION ACTIVITY | 1.2.4: Python basics.

1) What is the purpose of variables?

- ○ Store values for later use.
- ○

- Instruct the processor to execute an action.
- Automatically color text in the editor.

2) The code `20 * 40` is an expression.

- True
- False

3) How are most Python programs developed?

- Writing code in the interactive interpreter.
- Writing code in files.

4) Comments are required in a program.

- True
- False

zyDE 1.2.1: A first program.

Below is the zyBooks Development Environment (zyDE), a web-based programming pra environment. Click run to execute the program, then observe the output. Change 20 to a different number like 35 and click run again to see the different output.

Load default template...

Run

```
wage = 20
hours = 40
weeks = 52
salary = wage * hours * weeks

print('Salary is:', salary)

hours = 35
salary = wage * hours * weeks
print('New salary is:', salary)

```

1.3 Basic input and output

Basic text output

Printing output to a screen is a common programming task. This section describes basic output; later sections have more details.

The primary way to print output is to use the built-in function ***print()***. Ex: `print('hello world')`. Text enclosed in quotes is known as a ***string literal***. Text in string literals may have letters, numbers, spaces, or symbols like @ or #. Each use of `print()` starts on a new line.

A string literal can be surrounded by matching single or double quotes: `'Python rocks!'` or `"Python rocks!"`. Good practice is to use single quotes for shorter strings and double quotes for more complicated text or text that contains single quotes, like `print("Don't eat that!")`.

Figure 1.3.1: Printing text.

```
# Each print statement starts on a new line
print('Hello there.')
print('My name is...')
print('Carl?')
```

```
Hello there.
My name is...
Carl?
```

PARTICIPATION ACTIVITY 1.3.1: Basic text output.

1) Select the statement that prints the following: *Welcome!*

- print(Welcome!)
- print('Welcome!")
- print('Welcome!')

PARTICIPATION ACTIVITY 1.3.2: Basic text output.

1) Type a statement that prints the following: *Hello*

Check Show answer

CHALLENGE ACTIVITY 1.3.1: Output simple text.

Write the simplest statement that prints the following:

```
3 2 1 Go!
```

Note: Whitespace (blank spaces / blank lines) matters; make sure your whitespace *exactly* matches the expected output.

428154.2733946.qx3zqy7

```

''' Your solution goes here '''

```

Run

View your last submission

CHALLENGE ACTIVITY 1.3.2: Output an eight with asterisks.

Output the following figure with asterisks. Do not add spaces after the last character in each line.

```
*****
*   *
*****
*   *
*****
```


Note: Whitespace (blank spaces / blank lines) matters; make sure your whitespace *exactly* matches the expected output.

428154.2733946.qx3zqy7

```

''' Your solution goes here '''

```

Run

View your last submission

Keeping output on the same line

Each call to `print()` outputs on a new line. However, sometimes a programmer may want to keep output on the same line. A programmer can add `end=' '` inside of `print()` to keep the output of a subsequent print statement on the same line separated by a single space. Ex: `print('Hello', end=' ')`.

Figure 1.3.2: Printing text on the same row.

```
# Including end=' ' keeps output on same line
print('Hello there.', end=' ')
print('My name is...', end=' ')
print('Carl?')
```

```
Hello there. My name is... Carl?
```


PARTICIPATION ACTIVITY 1.3.3: Printing text on the same row.

1) Which pair of statements print output on the same line?

- print('Halt!')
  print('No access!')
- print('Halt!', end=' ')
  print('No access!')
- print(Halt!, end=' ')
  print(No Access!, end=' ')

Outputting a variable's value

The value of a variable can be printed out via: `print(variable_name)` (without quotes).

Figure 1.3.3: Printing the value of a variable.

```
wage = 20

print('Wage is', end=' ')
print(wage)  # print variable's value
print('Goodbye.')
```

```
Wage is 20
Goodbye.
```


PARTICIPATION ACTIVITY 1.3.4: Basic variable output.

1) Given the variable num_cars = 9, which statement prints 9?

- print(num_cars)
- print("num_cars")

PARTICIPATION ACTIVITY | 1.3.5: Basic variable output.

1) Write a statement that prints the value of the variable num_people.

Check Show answer

Outputting multiple items with one statement

Programmers commonly try to use a single print statement for each line of output by combining the printing of text, variable values, and new lines. A programmer can simply separate the items with commas, and each item in the output will be separated by a space. Combining string literals, variables, and new lines can improve program readability, because the program's code corresponds more closely to the program's printed output.

Figure 1.3.4: Printing multiple items using a single print statement.

```
wage = 20

print('Wage:', wage)  # Comma separates multiple
items
print('Goodbye.')
```

```
Wage: 20
Goodbye.
```

A common error is to forget the comma between items, as in `print('Name' user_name)`.

Newline characters

Output can be moved to the next line by printing `"\n"`, known as a ***newline character***. Ex: `print('1\n2\n3')` prints "1" on the first line, "2" on the second line, and "3" on the third line of output. `"\n"` consists of two characters, \ and n, but together are considered by the Python interpreter as a single character.

Figure 1.3.5: Printing using newline characters.

```
print('1\n2\n3')
```

```
1
2
3
```


`print()` always adds a newline character after the output automatically to move the next output to the next row, unless `end=' '` is provided to replace the newline character with a space (or some other character). An empty `print()` can be used to print only a newline.

Figure 1.3.6: printing without text.

```
print('123')
print()
print('abc')
```

```
123

abc
```

Any space, tab, or newline is called ***whitespace***.

NOTE: In a normal programming environment, program input is provided interactively and completed by pressing the enter key. The enter key press would insert a newline. Since zyBooks input is pre-entered, no enter key press can be inferred. Thus, activities that require pre-entered input may need extra newline characters or blank print statements in zyBooks, compared to other environments.

PARTICIPATION ACTIVITY 1.3.6: Output simulator.

The tool below supports a subset of Python, allowing for experimenting with print statements. The activity is marked as complete upon interacting with the tool (press Enter after entering the code).

The variables `country_population = 1344130000` and `country_name = 'China'` have been defined and can be used in the simulator.

Try printing the following output:

```
The population of China was 1344130000 in 2011.
```

Remember, commas can be used to output multiple items with a single print statement. Ex: `print('The person lives in, country_name, 'with their family.')` outputs

```
The person lives in China with their family.
```

```
print( "Change this string!" )
```

```
Change this string!
```


PARTICIPATION ACTIVITY | 1.3.7: Single print statement.

Assume variable age = 22, pet = "dog", and pet_name = "Gerald".

1) What is the output of `print('You are', age, 'years old.')`

Check Show answer

2) What is the output of `print(pet_name, 'the', pet, 'is', age)`

Check Show answer

CHALLENGE ACTIVITY | 1.3.3: Enter the output.

Type the program's output. Remember that print() outputs an ending newline, so press Enter or Return on your keyboard to indicate a newline in your answer.

428154.2733946.qx3zqy7

Start

Type the program's output

```
print('Ann is happy.')
```

```
Ann is happy.
```

1 2 3

Check Next

Basic input

Many useful programs allow a user to enter values, such as typing a number, a name, etc.

Reading input is achieved using the ***input()*** function. The statement `best_friend = input()` will read text entered by the user and the best_friend variable is assigned with the entered text. The function input() causes the interpreter to wait until the user has entered some text and has pushed the return key.

The input obtained by input() is any text that a user typed, including numbers, letters, or special characters like # or @. Such text in a computer program is called a ***string***.

A string simply represents a sequence of characters. For example, the string 'Hello' consists of the characters 'H', 'e', 'l', 'l', and 'o'. Similarly, the string '123' consists of the characters '1', '2', and '3'.

PARTICIPATION ACTIVITY 1.3.8: A program can get an input value from the keyboard.

Animation captions:

1. The input() statement gets an input value from the keyboard and puts that value into the best_friend variable.
2. best_friend's value can then be used in subsequent processing and outputs.

PARTICIPATION ACTIVITY 1.3.9: Reading user input.

1) Which statement reads a user-entered string into variable num_cars?

- `num_cars = input`
- `input() = num_cars`
- `num_cars = input()`

PARTICIPATION ACTIVITY 1.3.10: Reading user input.

1) Complete a statement that reads a user-entered string into variable my_var.

Check Show answer

Converting input types

The string '123' (with quotes) is fundamentally different from the integer 123 (without quotes). The '123' string is a sequence of the characters '1', '2', and '3' arranged in a certain order, whereas 123 represents the integer value one-hundred twenty-three. Strings and integers are each an example of a ***type***; a type determines how a value can behave. For example, integers can be divided by 2, but not strings (what sense would `"Hello" / 2` make?). Types are discussed in detail later on.

Reading from input always results in a string type. However, often a programmer wants to read in an integer, and then use that number in a calculation. If a string contains only numbers, like '123', then the ***int()*** function can be used to convert that string to the integer 123.

Figure 1.3.7: Using int() to convert strings to integers.

```
my_string = '123'
my_int =
int('123')

print(my_string)
print(my_int)
```

```
123
123
```

A programmer can combine input() and int() to read in a string from the user and then convert that string to an integer for use in a calculation.

Figure 1.3.8: Converting user input to integers.


```
Enter wage:
8
New wage: 18
```

```
print('Enter wage:', end='
')
wage = int(input())

new_wage = wage + 10
print('New wage:',
new_wage)
```


PARTICIPATION ACTIVITY 1.3.11: Converting user input to integers.

1) Type a statement that converts the string '15' to an integer and assigns my_var with the result.

Check Show answer

2) Complete the code so that new_var is equal to the entered number plus 5.

```
my_var = int(input())
new_var = 
```

Check Show answer

Input prompt

Adding a string inside the parentheses of input() displays a prompt to the user before waiting for input and is a useful shortcut to adding an additional print statement line.

Figure 1.3.9: Basic input example.


```
hours = 40
weeks = 52
hourly_wage = int(input('Enter hourly wage:
'))

print('Salary is', hourly_wage * hours *
weeks)
```

```
Enter hourly wage:
12
Salary is 24960
...
Enter hourly wage:
20
Salary is 41600
```

NOTE: The below tool requires input to be pre-entered. This is a current limitation of the web-based tool and atypical of conventional Python environments, where users enter input as the program runs. For conventional behavior, you may copy-paste the program into a local environment, such as IDLE.

zyDE 1.3.1: Basic input.

Run the program and observe the output. Change the input box value from 3 to another number, and run again.

Load default templ

```
human_years = int(input('Enter age of dog (in human years): '))
print()

dog_years = 7 * human_years

print(human_years, 'human years is about', end=' ')
print(dog_years, 'dog years.')



```

3

Run

CHALLENGE ACTIVITY 1.3.4: Read user input numbers and perform a calculation.

The following program reads in 2 numbers from input, assigns them to num1 and num2 respectively, and then outputs the sum of those numbers. Copy the code provided to see how this code is executed in an autograded system.

```
num1 = int(input())
num2 = int(input())
print(num1 + num2)
```

Note: Our autograder automatically runs your program several times, trying different input values each time to ensure your program works for any values. This program is tested twice, first with the inputs 5 and 10, and then with the inputs 6 and 3. See How to Use zyBooks for info on how our automated program grader works.

428154.2733946.qx3zqy7

```

''' Your solution goes here '''

```

Run

View your last submission

CHALLENGE ACTIVITY 1.3.5: Read user input and print to output.

Read three integers from user input **without a prompt**. Then, print the product of those integers. Ex: If input is 2 3 5, output is 30.

Note: Our system will run your program several times, automatically providing different input values each time, to ensure your program works for any input values. See How to Use zyBooks for info on how our automated program grader works.

428154.2733946.qx3zqy7

```
''' Your solution goes here '''
```


Run

View your last submission

CHALLENGE ACTIVITY

1.3.6: Output basics.

For activities with output like below, your output's whitespace (newlines or spaces) must match exactly. See this note.

428154.2733946.qx3zqy7

Start

Write code that outputs the following. End with a newline. Remember that print() automatically adds a newline.

```
This dinner will be awesome.
```

```
''' Your code goes here '''
```

1 2 3 4 5 6

Check Next Show solution

1.4 Errors

Syntax errors

As soon as a person begins trying to program, that person will make mistakes. One kind of mistake, known as a ***syntax error***, is to violate a programming language's rules on how symbols can be combined to create a program. An example is putting multiple prints on the same line.

The interpreter will generate a message when encountering a syntax error. The error message will report the number of the offending line, in this case 7, allowing the programmer to go back and fix the problem. Sometimes error messages can be confusing, or not particularly helpful. Below, the message "invalid syntax" is not very precise, but is the best information that the interpreter is able to report. With enough practice a programmer becomes familiar with common errors and is able to avoid them, avoiding headaches later.

Note that syntax errors are found *before* the program is ever run by the interpreter. In the example below, none of the prints prior to the error is in the output.

Figure 1.4.1: A program with a syntax error.

```
print('Current salary is', end='
')
print(45000)

print('Enter new salary:', end='
')
new_sal = int(input())

print(new_sal) print(user_num)
```

```
File "<main.py>", line 7
    print(new_sal)
print(user_num)
             ^
SyntaxError: invalid syntax
```

PARTICIPATION ACTIVITY 1.4.1: Syntax errors.

Find the syntax errors. Assume variable num_dogs exists.

1) print(num_dogs).

- Error
- No Error

2) print("Dogs: " num_dogs)

- Error
- No Error

3) print('Woof!")

- Error
- No Error

4) print(Woof!)

- Error
- No Error

5) print("Hello + friend!")

- Error
- No Error

PARTICIPATION ACTIVITY 1.4.2: Common syntax errors.

Treat the following lines of code as a continuous program. Find and click on the 3 syntax errors.

1)

```
triangle_base = 0    # Triangle base (cm)
triangle_height = 0  # Triangle height (cm)
triangle_area = 0
       # Triangle area (cm**2)

print('Enter triangle base (cm): ')
triangle_base = int(input()
```


2)

```
print('Enter triangle height (cm): ')
```

triangle_height = int(input())

Calculate triangle area

triangle_area = (triangle_base * triangle_height) / 2

Print out the triangle base, height, and area

3) print('Triangle area = (', end=' ')

print(triangle_base, end=' ')

print(*, end=' ')

print(triangle_height, end=' ')

print(') / 2 = ', end=' ')
print(triangle_area, end=' ')
print('cm**2')

Good coding practice

New programmers will commonly write programs with many syntax errors, leading to many frustrating error messages. To avoid continually encountering error messages, a good practice is to execute the code frequently, writing perhaps a few (3–5) lines of code and then fixing errors, then writing a few more lines and running again and fixing errors, and so on. Experienced programmers may write more lines of code each time, but typically still run and test syntax frequently.

PARTICIPATION ACTIVITY 1.4.3: Run code frequently to avoid many errors.

Animation captions:

1. Writing many lines of code without compiling is bad practice.
2. New programmers should compile their program after every couple of lines.

PARTICIPATION ACTIVITY 1.4.4: Testing for syntax errors.

1) Experienced programmers write an entire program before running and testing the code.

- True
- False

Runtime errors

The Python interpreter is able to detect syntax errors when the program is initially loaded, prior to actually executing any of the statements in the code. However, just because the program loads and executes does not mean that the program is correct. The program may have another kind of error called a ***runtime error***, wherein a program's syntax is correct but the program attempts an impossible operation, such as dividing by zero or multiplying strings together (like 'Hello' * 'ABC').

A runtime error halts the execution of the program. Abrupt and unintended termination of a program is often called a ***crash*** of the program.

Consider the below program that begins executing, prints the salary, and then waits for the user to enter an integer value. The int() statement expects a number to be entered, but gets the text 'Henry' instead.

Figure 1.4.2: Runtime errors can crash the program.

The program crashes because the user enters 'Henry' instead of an integer value.

```
print('Salary is', end=' ')
print(20 * 40 * 50)

print('Enter integer: ', end=' ')
user_num = int(input())
print(user_num)
```

```
Salary is 40000
Enter integer:  Henry
Traceback (most recent call last):
 File "<stdin>", line 5, in <module>
ValueError: invalid literal for int() with base 10:
'Henry'
```

Runtime errors are categorized into types that describe the sort of error that has occurred. Above, a *ValueError* occurred, indicating that the wrong sort of value was passed into the int() function. Other examples include a NameError and a TypeError, both described in the table below.

Common error types

Table 1.4.1: Common error types.

Error type	Description

SyntaxError	The program contains invalid code that cannot be understood.
IndentationError	The lines of the program are not properly indented.
ValueError	An invalid value is used – can occur if giving letters to int().
NameError	The program tries to use a variable that does not exist.
TypeError	An operation uses incorrect types – can occur if adding an integer to a string.

PARTICIPATION ACTIVITY 1.4.5: Match the lines of code with the error type that they produce.

Match the following lines of code with the correct error type. Assume that no variables already exist.

IndentationError SyntaxError TypeError NameError ValueError

	lyric = 99 + " bottles of pop on the wall"
	print("Friday, Friday")
	int("Thursday")
	day_of_the_week = Friday
	print('Today is Monday")

Reset

Logic errors

Some errors may be subtle enough to silently misbehave, instead of causing a runtime error and a crash. An example might be if a programmer accidentally typed "2 * 4" rather than "2 * 40" – the program would load correctly, but would not behave as intended. Such an error is known as a ***logic error***, because the program is logically flawed. A logic error is often called a ***bug***.

Figure 1.4.3: The programmer made a mistake that happens to be correct

syntax, but has a different meaning.

The below program attempts to calculate a 5% raise for an employee's salary. The programmer made a mistake by assigning raise_percentage with 5, instead of 0.05, thus giving a happy employee a 500% raise.

```
current_salary = int(input('Enter current salary:'))

raise_percentage = 5  # Logic error gives a 500% raise instead of 5%.
new_salary = current_salary + (current_salary * raise_percentage)
print('New salary:', new_salary)
```



```
Enter current salary:
10000
New salary: 60000
```

The programmer clearly made an error, but the code is actually correct syntax – it just has a different meaning than was intended. So the interpreter will not generate an error message, but the program's output is not what the programmer expects – the new computed salary is much too high. These mistakes can be very hard to debug. Paying careful attention and running code after writing just a few lines can help avoid mistakes.

zyDE 1.4.1: Fix the bug.

Click run to execute the program and note the incorrect program output. Fix the bug in t program.

Load default template...

Run

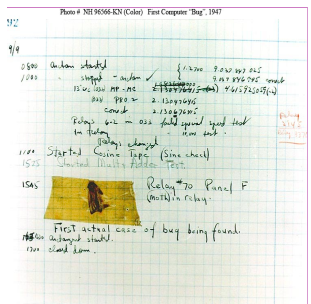
```
num_beans = 500
num_jars = 3
total_beans = 0

print(num_beans, 'beans in', end=' ')
print(num_jars, 'jars yields', end=' ')
total_beans = num_beans * num_jars
print('total_beans', 'total')

```

Figure 1.4.4: The first bug.

A sidenote: The term "bug" to describe a runtime error was popularized in 1947. That year, engineers working with pioneering computer scientist Grace Hopper discovered their program on a Harvard University Mark II computer was not working because a moth was stuck in one of the relays (a type of mechanical switch). They taped the bug into their engineering log book, which is still preserved today (http://en.wikipedia.org/wiki/Computer_bug).



CHALLENGE ACTIVITY 1.4.1: Basic syntax errors.

Retype the statements, correcting the syntax error in each print statement.

```
print('Predictions are hard.")
print(Especially about the future.)
user_num = 5
print('user_num is:' user_num)
```

428154.2733946.qx3zqy7

```
''' Your solution goes here '''

```

Run

View your last submission

1.5 Additional practice: Output art

The following is a sample programming lab activity; not all classes using a zyBook require students to fully complete this activity. No auto-checking is performed. Users planning to fully complete this program may consider first developing their code in a separate programming environment.

Pictures made entirely from keyboard characters are known as ASCII art. ASCII art can be quite complex, fun to make, and enjoyable to view. Take a look at Wikipedia: ASCII art for examples. Doing a web search for "ASCII art (some item)" can find ASCII art versions of an item; e.g., searching for "ASCII art cat" turns up thousands of examples of cats, most much more clever than the cat below.

The following program outputs a simple triangle.

Figure 1.5.1: Output art: Printing a triangle.

```
print('  *  ')
print(' *** ')
print('*****')
```

```
  *
 ***
*****
```


zyDE 1.5.1: Create ASCII art.

Create different versions of the below programs. First run the code, then alter the code t
the desired output.

Print a tree by adding a base under a 4-level triangle:

```
   *
  ***
 *****
*******
  ***
```

Load default template...

Run

```
print('   *   ')
print('  ***  ')
print(' ***** ')
print('*******')
```


zyDE 1.5.2: Create ASCII art.

Complete the cat drawing below. Note the '\' character is actually displayed by printing t character sequence '\\'.

```
/\   /\
  o o
 =   =
  ---
```

Load default template...

Run

```
print('/\\      /\\')
print('  o    ')
```

zyDE 1.5.3: Create ASCII art.

Be creative: Print something you'd like to see that is more impressive than the previous programs.

No template provided

Run

```
1
```

1.6 Development environment

IDEs

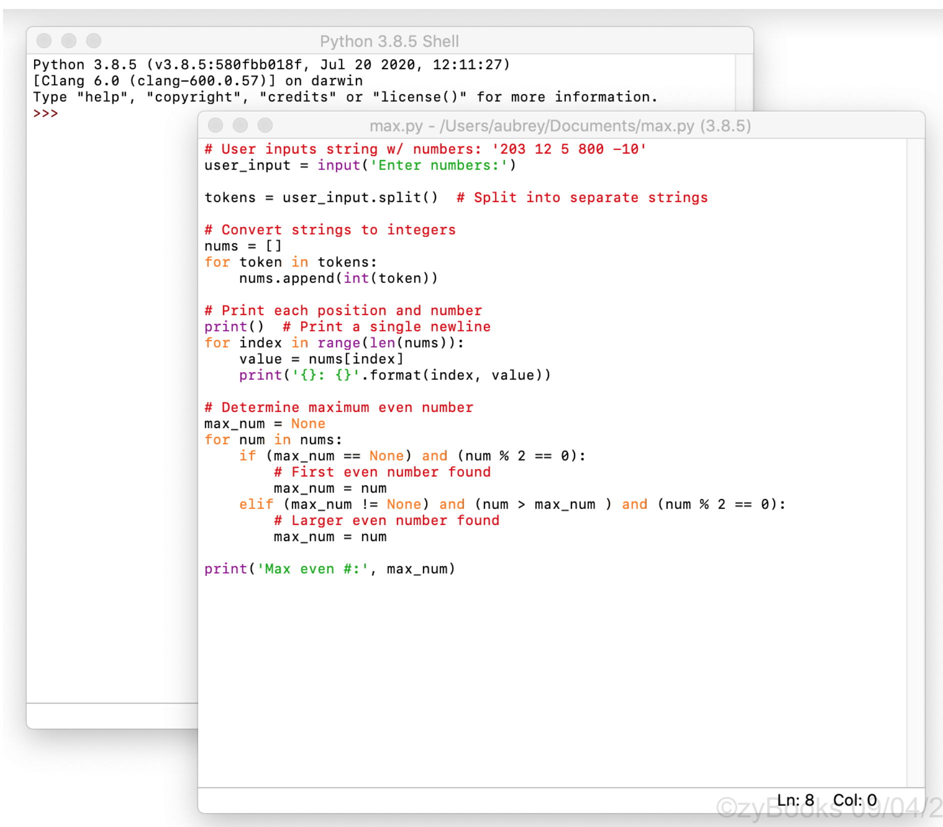
This web material embeds a Python interpreter so that the reader may experiment with Python programming. However, for normal development, a programmer installs Python as an application on a local computer. Macintosh and Linux operating systems usually include Python, while Windows does not. Programmers can download the latest version of Python for free from https://python.org.

Code development is usually done with an ***integrated development environment***, or ***IDE***. There are various IDEs that can be found online; some of the most popular are listed below.

- IDLE is the official Python IDE that is distributed with the installation of Python from https://python.org. IDLE provides a basic environment for editing and running programs.
- PyDev (http://pydev.org) is a plugin for the popular Eclipse program. PyDev includes extra features such as code completion, spell checking, and a debugger that can be useful tools while programming.
- For learning purposes, web-based tools like CodePad (http://www.codepad.co) or Repl (http://www.repl.it) are useful.

There are many other editors available—some of which are free, while others require a fee or subscription. Finding the right IDE is sometimes like finding a pair of jeans that fits just right—try a Google search for "Python IDE" and explore the options.

Figure 1.6.1: IDLE environment for coding and running Python.



PARTICIPATION ACTIVITY 1.6.1: Development environment basics.

1) Python comes pre-installed on Windows machines.

True

False

2) Python code can be written in a simple text editor, such as Notepad (Windows).

True

False

1.7 Computers and programs (general)

Figure 1.7.1: Looking under the hood of a car.

Source: zyBooks

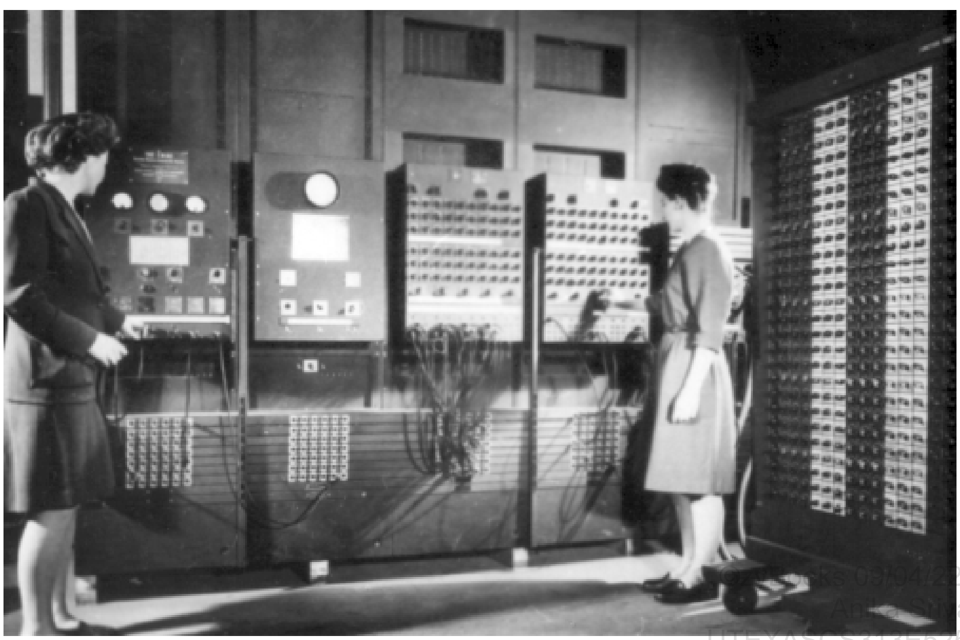
Just as knowing how a car works "under-the-hood" has benefits to a car owner, knowing how a computer works under-the-hood has benefits to a programmer. This section provides a very brief introduction.

Switches

When people in the 1800s began using electricity for lights and machines, they created switches to turn objects on and off. A *switch* controls whether or not electricity flows through a wire. In the early 1900s, people created special switches that could be controlled electronically, rather than by a person moving the switch up or down. In an electronically controlled switch, a positive voltage at the control input allows electricity to flow, while a zero voltage prevents the flow. Such switches were useful, for example, in routing telephone calls. Engineers soon realized they could use electronically controlled switches to perform simple calculations. The engineers treated a positive voltage as a "1" and a zero voltage as a "0". 0s and 1s are known as ***bits*** (*b*inary dig*its*). They built connections of switches, known as *circuits*, to perform calculations such as multiplying two numbers.

PARTICIPATION ACTIVITY | 1.7.1: A bit is either 1 or 0, like a light switch is either on or off (click the switch).

Figure 1.7.2: Early computer made from thousands of switches.



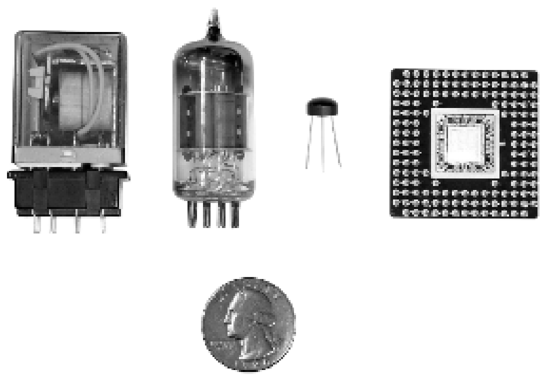
Source: ENIAC computer (U. S. Army Photo / Public domain)

These circuits became increasingly complex, leading to the first electronic computers in the 1930s and 1940s, consisting of about ten thousand electronic switches and typically occupying entire rooms as in the above figure. Early computers performed thousands of calculations per second, such as calculating tables of ballistic trajectories.

Processors and memory

To support different calculations, circuits called ***processors*** were created to process (aka *execute*) a list of desired calculations, each calculation called an ***instruction***. The instructions were specified by configuring external switches, as in the figure below. Processors used to take up entire rooms, but today fit on a chip about the size of a postage stamp, containing millions or even billions of switches.

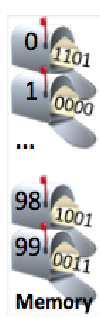
Figure 1.7.3: As switches shrunk, so did computers. The computer processor chip on the right has millions of switches.



Source: zyBooks

Instructions are stored in a memory. A ***memory*** is a circuit that can store 0s and 1s in each of a series of thousands of addressed locations, like a series of addressed mailboxes that each can store an envelope (the 0s and 1s). Instructions operate on data, which is also stored in memory locations as 0s and 1s.

Figure 1.7.4: Memory.



Thus, a computer is basically a processor interacting with a memory. In the following example, a computer's processor executes program instructions stored in memory, also using the memory to store temporary results. The example program converts an hourly wage ($20/hr) into an annual salary by multiplying by 40 (hours/week) and then by 52 (weeks/year), outputting the final result to the screen.

PARTICIPATION ACTIVITY | 1.7.2: Computer processor and memory.

Animation content:

undefined

Animation captions:

1. The processor computes data, while the memory stores data (and instructions).
2. Previously computed data can be read from memory.
3. Data can be output to the screen.

The arrangement is akin to a chef (processor) who executes instructions of a recipe (program), where each instruction modifies ingredients (data), with the recipe and ingredients kept on a nearby counter (memory).

Instructions

Below are some sample types of instructions that a processor might be able to execute, where *X*, *Y*, *Z*, and *num* are each an integer.

Table 1.7.1: Sample processor instructions.

Add X, #num, Y	Adds data in memory location *X* to the number *num*, storing result in location *Y*
Sub X, #num, Y	Subtracts *num* from data in location *X*, storing result in location *Y*
Mul X, #num, Y	Multiplies data in location *X* by *num*, storing result in location *Y*
Div X, #num, Y	Divides data in location *X* by *num*, storing result in location *Y*
Jmp Z	Tells the processor that the next instruction to execute is in memory location *Z*

For example, the instruction "Mul 97, #9, 98" would multiply the data in memory location 97 by the number 9, storing the result into memory location 98. So if the data in location 97 were 20, then the instruction would multiply 20 by 9, storing the result 180 into location 98. That instruction would actually be stored in memory as 0s and 1s, such as "011 1100001 001001 1100010" where 011 specifies a multiply instruction, and 1100001, 001001, and 1100010 represent 97, 9, and 98 (as described previously). The following animation illustrates the storage of instructions and data in memory for a program that computes F = (9*C)/5 + 32, where C is memory location 97 and F is memory location 99.

PARTICIPATION ACTIVITY | 1.7.3: Memory stores instructions and data as 0s and 1s.

Animation captions:

1. Memory stores instructions and data as 0s and 1s.
2. This zyBook will commonly draw the memory with the corresponding instructions and data to improve readability.

The programmer-created sequence of instructions is called a ***program***, ***application***, or just ***app***.

When powered on, the processor starts by executing the instruction at location 0, then location 1, then location 2, etc. The above program performs the calculation over and over again. If location 97 is connected to external switches and location 99 to external lights, then a computer user (like the women in the above picture) could set the switches to represent a particular Celsius number, and the computer would automatically output the Fahrenheit number using the lights.

PARTICIPATION ACTIVITY | 1.7.4: Processor executing instructions.

Animation captions:

1. The processor starts by executing the instruction at location 0.
2. The processor next executes the instruction at location 1, then location 2. 'Next' keeps track of the location of the next instruction.
3. The Jmp instruction indicates that the next instruction to be executed is at location 0, so Next is assigned with 0.
4. The processor executes the instruction at location 0, performing the same sequence of instructions over and over again.

PARTICIPATION ACTIVITY | 1.7.5: Computer basics.

1) A bit can only have the value of 0 or 1.

True

False

2) Switches have gotten larger over the years.

True

False

3) Memory stores bits.

True

False

4) The computer inside a modern smartphone would have been huge in 1960.

True

False

5) A processor executes instructions such as Add 200, #9, 201, represented as 0s and 1s.

True

False

Writing computer programs

In the 1940s, programmers originally wrote each instruction using 0s and 1s, such as "001 1100001 001001 1100010". Instructions represented as 0s and 1s are known as ***machine instructions***, and a sequence of machine instructions together form an ***executable program*** (sometimes just called an *executable*). Because 0s and 1s are hard to comprehend, programmers soon created programs called *assemblers* to automatically translate human readable instructions, such as "Mul 97, #9, 98", known as ***assembly*** language instructions, into machine instructions. The assembler program thus helped programmers write more complex programs.

In the 1960s and 1970s, programmers created ***high-level languages*** to support programming using formulas or algorithms, so a programmer could write a formula like `F = (9 /5 ) * C + 32`. Early high-level languages included *FORTRAN* (for "Formula Translator") or *ALGOL* (for "Algorithmic Language"), which were more closely related to how humans thought than were machine or assembly instructions.

To support high-level languages, programmers created ***compilers***, which are programs that automatically translate high-level language programs into executable programs.

PARTICIPATION ACTIVITY | 1.7.6: Program compilation and execution.

Animation content:

`undefined`

Animation captions:

1. A programmer writes a high-level program.
2. The programmer runs a compiler, which converts the high-level program into an executable program.
3. Users can then run the executable.

PARTICIPATION ACTIVITY | 1.7.7: Programs.

Machine instruction **Assembly language** **Compiler** **Application**

	Translates a high-level language program into low-level machine instructions.
	Another word for program.
	A series of 0s and 1s, stored in memory, that tells a processor to carry out a particular operation like a multiplication.
	Human-readable processor instructions

Reset

Note (mostly for instructors): Why introduce machine-level instructions in a high-level language book? Because a basic understanding of how a computer executes programs can help students master high-level language programming. The concept of sequential execution (one instruction at a time) can be clearly made with machine instructions. Even more importantly, the concept of each instruction operating on data in memory can be clearly demonstrated. Knowing these concepts can help students understand the idea of assignment (x = x + 1) as distinct from equality, why x = y; y = x does not perform a swap, what a pointer or variable address is, and much more.

1.8 Computer tour

The term *computer* has changed meaning over the years. The term originally referred to a person that performed computations by hand, akin to an accountant ("We need to hire a computer"). In the 1940s/1950s, the term began to refer to large machines like in the earlier photo. In the 1970s/1980s, the term expanded to also refer to smaller home/office computers known as personal computers or PCs ("personal" because the computer wasn't shared among multiple users like the large ones) and to portable/laptop computers. In the 2000s/2010s, the term may also cover other computing devices like pads, book readers, and smartphones. The term computer even refers to computing devices embedded inside other electronic devices such as medical equipment, automobiles, aircraft, consumer electronics, and military systems.

In the early days of computing, the physical equipment was prone to failures. As equipment became more stable and as programs became larger, the term "software" became popular to distinguish a computer's programs from the "hardware" on which they ran.

A computer typically consists of several components (see animation below):

- ***Input/output devices***: A ***screen*** (or monitor) displays items to a user. The above examples displayed textual items, but today's computers display graphical items too. A ***keyboard*** allows a user to provide input to the computer, typically accompanied by a *mouse* for graphical displays. Keyboards and mice are increasingly being replaced by *touchscreens*. Other devices provide additional input and output means, such as microphones, speakers, printers, and USB interfaces. I/O devices are commonly called *peripherals*.
- ***Storage***: A ***disk*** (aka *hard drive*) stores files and other data, such as program files, song/movie files, or office documents. Disks are *non-volatile*, meaning they maintain their contents even when powered off. They do so by orienting magnetic particles in a 0 or 1 position. The disk spins under a head that pulses electricity at just the right times to orient specific particles (you can sometimes hear the disk spin and the head clicking as the head moves). New *flash* storage devices store 0s and 1s in a non-volatile memory rather than disk, by tunneling electrons into special circuits on the memory's chip, and removing them with a "flash" of electricity that draws the electrons back out.
- ***Memory***: ***RAM*** (random-access memory) temporarily holds data read from storage and is designed such that any address can be accessed much faster than disk, in just a few clock ticks (see below) rather than hundreds of ticks. The "random access" term comes from being able to access any memory location quickly and in arbitrary order, without having to spin a disk to get a proper location under a head. RAM is costlier per bit than disk, due to RAM's higher speed. RAM

chips typically appear on a printed circuit board along with a processor chip. RAM is volatile, losing its contents when powered off. Memory size is typically listed in bits, or in bytes where a ***byte*** is 8 bits. Common sizes involve megabytes (million bytes), gigabytes (billion bytes), or terabytes (trillion bytes).

- ***Processor***: The ***processor*** runs the computer's programs, reading and executing instructions from memory, performing operations, and reading/writing data from/to memory. When powered on, the processor starts executing the program whose first instruction is (typically) at memory location 0. That program is commonly called the BIOS (basic input/output system), which sets up the computer's basic peripherals. The processor then begins executing a program called an *operating system (OS)*. The ***operating system*** allows a user to run other programs and interfaces with the many other peripherals. Processors are also called *CPUs* (central processing unit) or *microprocessors* (a term introduced when processors began fitting on a single chip, the "micro" suggesting small). Because speed is so important, a processor may contain a small amount of RAM on its own chip, called ***cache*** memory, accessible in one clock tick rather than several, for maintaining a copy of the most-used instructions/data.
- ***Clock***: A processor's instructions execute at a rate governed by the processor's ***clock***, which ticks at a specific frequency. Processors have clocks that tick at rates such as 1 MHz (1 million ticks/second) for an inexpensive processor ($1) like those found in a microwave oven or washing machine, to 1 GHz (1 billion ticks/second) for costlier ($10-$100) processors like those found in mobile phones and desktop computers. Executing about 1 instruction per clock tick, processors thus execute millions or billions of instructions per second.

Computers typically run multiple programs simultaneously, such as a web browser, an office application, and a photo editing program. The operating system actually runs a little of program A, then a little of program B, etc., switching between programs thousands of times a second.

PARTICIPATION ACTIVITY 1.8.1: Some computer components.

Animation captions:

1. A disk is able to store Terabytes of data and may contain various programs such as ProgA, ProgB, Doc1, Doc2, and OS. The memory is able to store gigabytes of data. User runs ProgA. The disk spins and the head loads ProgA from the disk, storing the contents into memory.
2. The OS runs ProgB. The disk spins and the head loads ProgB from the disk, storing the contents into memory.
3. The OS lets ProgA run again. ProgA is already in memory so there is no need to read ProgA from the disk.

After computers were first invented and occupied entire rooms, engineers created smaller switches called ***transistors***, which in 1958 were integrated onto a single chip called an ***integrated circuit*** or ***IC***. Engineers continued to find ways to make smaller transistors, leading to what is known as ***Moore's Law***: The doubling of IC capacity roughly every 18 months, which continues today.[1] By 1971, Intel produced the first single-IC processor named the 4004, called a *microprocessor* ("micro" suggesting small), having 2300 transistors. New more-powerful microprocessors appeared every few years, and by 2012, a single IC had several *billion* transistors containing multiple processors (each called a *core*).

PARTICIPATION ACTIVITY | 1.8.2: Computer components.

Moore's Law | Operating system | Disk | Clock | Cache | RAM

	Manages programs and interfaces with peripherals.
	Non-volatile storage with slower access.
	Volatile storage with faster access usually located off processor chip.
	Relatively small, volatile storage with fastest access located on processor chip.
	Measures the speed at which a processor executes instructions.
	The doubling of IC capacity roughly every 18 months.

Reset

A sidenote: A common way to make a PC faster is to add more RAM. A processor spends much of its time moving instructions/data between memory and storage, because not all of a program's instructions/data may fit in memory—akin to a chef that spends most of his/her time walking back and forth between a stove and pantry. Just as adding a larger table next to the stove allows more ingredients to be kept close by, a larger memory allows more instructions/data to be kept close to the processor. Moore's Law results in RAM being cheaper a few years after buying a PC, so adding RAM to a several-year-old PC can yield good speedups for little cost.

Exploring further:

- Link: What's inside a computer (HowStuffWorks.com).

- "How Microprocessors Work" from howstuffworks.com.

(*1) Moore actually said every 2 years. And the actual trend has varied from 18 months. The key is that doubling occurs roughly every couple years, causing enormous improvements over time. Wikipedia: Moore's Law.

1.9 Language history

Scripting languages and Python

As computing evolved throughout the 1960s and 1970s, programmers began creating ***scripting languages*** to execute programs without the need for compilation. A ***script*** is a program whose instructions are executed by another program called an ***interpreter***. Interpreted execution is slower due to requiring multiple interpreter instructions to execute one script instruction, but has advantages including avoiding the compilation step during programming, and being able to run the same script on different processors as long as each processor has an interpreter installed.

In the late 1980s, Guido van Rossum began creating a scripting language called ***Python*** and an accompanying interpreter. He derived Python from an existing language called ABC. The name Python came from Guido being a fan of the TV show Monty Python. The goals for the language included simplicity and readability, while providing as much power and flexibility as other scripting languages like Perl.

Python 1.0 was released in 1994 with support for some functional programming constructs derived from Lisp. Python 2.0 was released in 2000 and introduced automatic memory management (garbage collection, described elsewhere) and features from Haskell and other languages. Python 2 officially reached "end-of-life" (no more fixes or support) in 2020 (Source: Python.org). Python 3.0 was released in 2008 to rectify various language design issues. Python 2.7 still remained popular for years, due largely to third-party libraries supporting only Python 2.7. Python 2.7 programs cannot run on Python 3.0 or later interpreters, i.e., Python 3.0 is not ***backwards compatible***. However, Python 3.x versions have become widely used as new projects adopted the version.

Python is an ***open-source*** language, meaning the community of users participate in defining the language and creating new interpreters, and is supported by a large community of programmers. A January 2022 survey that measured the popularity of various programming languages found that Python (13.58%) is the most popular language, and Python gained the most popularity of any language in 2021 (source: www.tiobe.com/tiobe-index/).

PARTICIPATION ACTIVITY | 1.9.1: Python background.

1) Python was first implemented in 1960.

- True
- False

2) Python is a high-level language that excels at creating exceptionally fast-executing programs.

- True
- False

3) A major drawback of Python is that Python code is more difficult to read than code in most other programming languages.

- True
- False

1.10 Why whitespace matters

Whitespace and precise formatting

For program output, ***whitespace*** is any blank space or newline. Most coding activities strictly require a student program's output to exactly match the expected output, including whitespace. Students learning programming often complain:

"My program is correct, but the system is complaining about output whitespace. "

However, correctness often includes output being formatted correctly.

PARTICIPATION ACTIVITY 1.10.1: Precisely formatting a meeting invite.

Animation content:

```
undefined
```

Animation captions:

1. This program for online meetings not only does computations like scheduling and creating a unique meeting ID, but also outputs text formatted neatly for a calendar event.
2. A calendar program may append more text after the meeting invitation text.
3. The programmer of the invitation on the right wasn't careful with whitespace. "Join meeting" is buried, the link is hard to see, and the "Phone" text is dangling at a line's end.
4. The programmer also didn't end with a newline, causing subsequent text to appear at the end of a line, and even wrap to the next line. This output looks unprofessional.

PARTICIPATION ACTIVITY 1.10.2: Program correctness includes correctly-formatted output.

Consider the example above.

1) The programmer on the left intentionally inserted a newline in the first sentence, namely "Kia Smith ... video meeting". Why?

- Probably a mistake
- So the text appears less jagged
- To provide some randomness to the output

2) The programmer on the right did not end the first sentence with a newline. What effect did that omission have?

- "Join meeting" appears on the same line
- No effect

3) The programmer on the left neatly formatted the link, the "Phone:" text, and phone numbers. What did the programmer on the right do?

- Also neatly formatted those items
- Output those items without neatly formatting

4) On the right, why did the "Reminder..." text appear on the same line as the separator text "------"?

- Because programs behave

erratically

- Because the programmer didn't end the output with a newline

5) Whitespace _____ important in program output.

- is
- is not

Programming is all about precision

Programming is all about *precision*. Programs must be created precisely to run correctly. Ex:

- = and == have different meanings.
- Using i where j was meant can yield a hard-to-find bug.
- Not considering that n could be 0 in sum/n can cause a program to fail entirely in rare but not insignificant cases.
- Counting from i being 0 to i < 10 vs. i <= 10 can mean the difference between correct output and a program outputting garbage.

In programming, every little detail counts. Programmers must get in a mindset of paying extreme *attention to detail*.

Thus, another reason for caring about whitespace in program output is to help new programmers get into a "precision" mindset when programming. Paying careful attention to details like whitespace instructions, carefully examining feedback regarding whitespace differences, and then modifying a program to exactly match expected whitespace is an exercise in strengthening attention to detail. Such attention can lead programmers to make fewer mistakes when creating programs, thus spending less time debugging, and instead creating programs that work correctly.

PARTICIPATION ACTIVITY 1.10.3: Thinking precisely, and attention to detail.

Programmers benefit from having a mindset of thinking precisely and paying attention to details. The following questions emphasize attention to detail. See if you can get all of the questions correct on the first try.

1) How many times is the letter F (any case) in the following?
If Fred is from a part of France, then of course Fred's French is good.

Check Show answer

2) How many differences are in these two lines?
Printing A linE is done using println
Printing A linE is done using print1n

Check Show answer

3) How many typos are in the following?
Keep calmn and cary on.

Check Show answer

4) If I and E are adjacent, I should come before E, except after C (where E should come before I). How many violations are in the following?
BEIL CEIL ZIEL YIEIK TREIL

Check Show answer

5) A password must start with a letter, be at least 6 characters long, include a number, not include any whitespace, and include a special symbol. How many of the following passwords are valid?
hello goodbye Maker1 dog!three Oops_again 1augh#3

Check Show answer

Check Show answer

Programmer attention to details

The focus needed to answer the above correctly on the first try is the kind of focus needed to write correct programs. Due to this fact, some employers give "attention to detail" tests to people applying for programming positions. See for example this test, or this article discussing the issue. Or, just web search for "programmer attention to details" for more such tests and articles.

1.11 Python example: Salary calculation

This section contains a very basic example for starters; the examples increase in size and complexity in later sections.

zyDE 1.11.1: Executing Python code using the interpreter.

The following program calculates yearly and monthly salary given an hourly wage. The program assumes 40 work hours per week and 50 work weeks per year.

1. Insert the correct number in the code below to print a monthly salary. Then run th program. The monthly salary should be 3333.333...

NOTE: This section does not have any activity to be recorded as completed, and thus dc count towards a student's activity completion percentages. The example is included by popular request of students, who often ask for more examples -- and research indeed s that students learn a lot by studying and tinkering with examples.

Load default template...

Run


```
hourly_wage = 20

print('Annual salary is: ')
print(hourly_wage * 40 * 50)
print()

print('Monthly salary is: ')
print(((hourly_wage * 40 * 50) / 1))
print()
```

```
# FIXME: The above is wrong. Change
#        the 1 so that the statement
#        outputs monthly salary.
```