2.1 Programming introduction

A recipe consists of *instructions* that a chef executes, like adding eggs or stirring ingredients. Likewise, a ***computer program*** consists of instructions that a computer executes (or *runs*), like multiplying numbers or printing a number to a screen.

Figure 2.1.1: A program is like a recipe.

Bake chocolate chip cookies:

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

PARTICIPATION ACTIVITY 2.1.1: A first computer program.

Run the program and observe the output. Click and drag the instructions to change the order of the instructions, and run the program again. Not required (points are awarded just for interacting), but can you make the program output a value greater than 500? How about greater than 1000?

Run program

```
m = 5

print m
```

```
m = m * 2
print m
```

```
m = m * m
print m
```

```
m = m + 15
print m
```

m:

PARTICIPATION ACTIVITY 2.1.2: Instructions.

Select the instruction that achieves the desired goal.

1) **Make lemonade:**

- Fill jug with water
- Add lemon juice
- _____
- Stir

- ○ Add salt
- ○ Add water
- ○ Add sugar

2) **Wash a car:**

- Fill bucket with soapy water
- Dip towel in bucket
- Wipe car with towel
- _____

- ○ Rinse car with hose
- ○ Add water to bucket
- ○ Add sugar to bucket

3) **Wash hair:**

- Rinse hair with water
- While hair isn't squeaky clean, repeat:
  - _____
  - Work shampoo throughout hair
  - Rinse hair with water

- ○ Rinse hair with water
- ○ Add shampoo to hair
- ○ Sing

4) **Compute the area of a triangle:**

- Determine the base
- Determine the height
- Compute base times height
- _____

- Multiply the previous answer by 2
- Add 2 to the previous answer
- Divide the previous answer by 2

2.2 Computational thinking

Imagine that a chef creates a new dish that she wants to share. The chef writes a recipe that consists of the most basic steps required to create a copy of the dish: chop one onion, dice two tomatoes, mix with lettuce, etc. ***Task decomposition*** means to reduce a complex task into simpler basic steps, making the whole task easier to solve. Like a new dish, a computer program is generally created by decomposing a task into simpler basic steps, and then instructing a computer to perform those steps in a certain order.

Table 2.2.1: Examples of task decomposition.

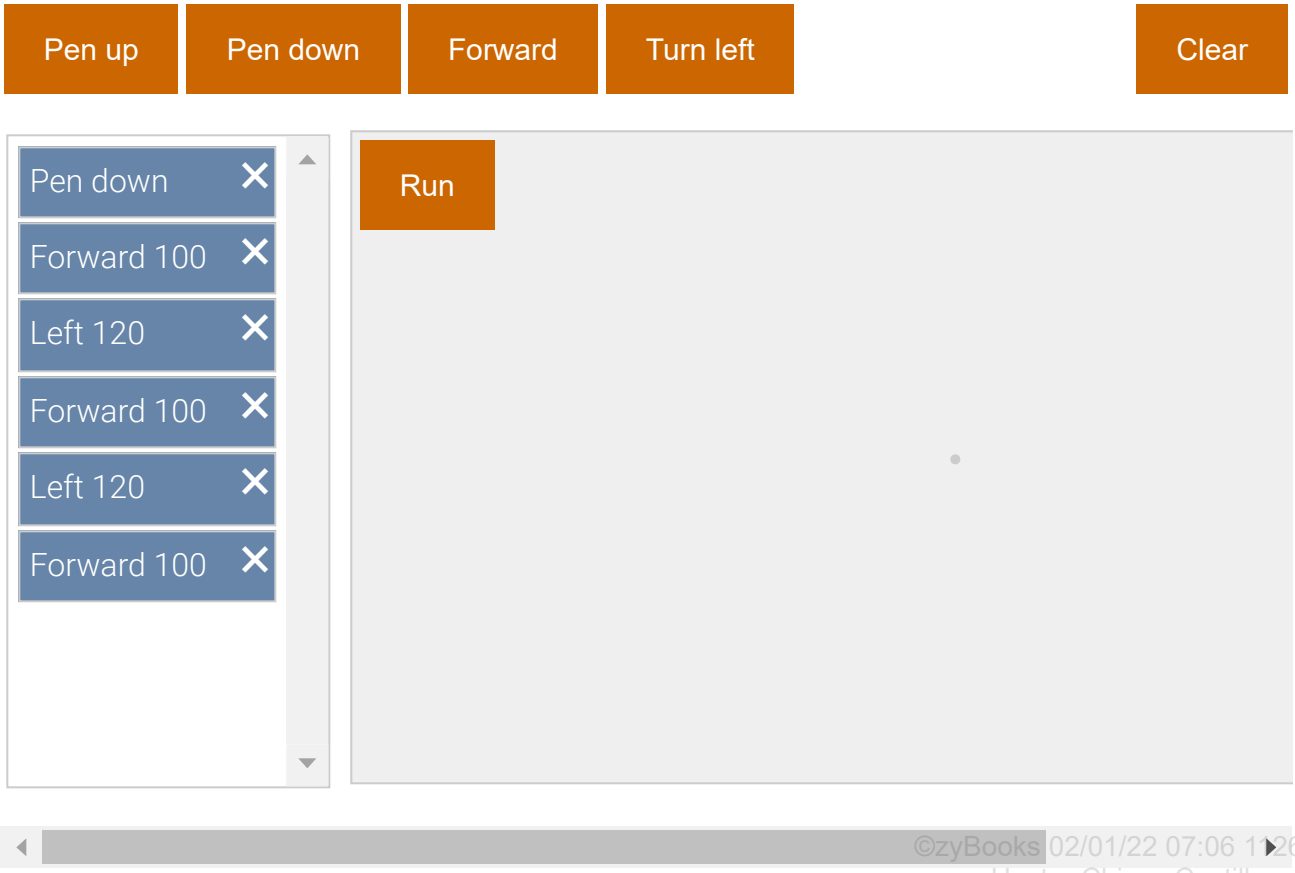
Task description	The task decomposed into group of simpler problems.
Fight crime	1. Put on cape 2. Put on mask 3. Get in crime-fighting vehicle 4. Arrest villain 5. Return to hideout
Vacuum room	1. Get vacuum from closet 2. Plug in vacuum to power outlet 3. Lower vacuum handle 4. Roll forwards 5. Roll backwards 6. Roll forwards 7. ... 8. Unplug vacuum 9. Store in closet

PARTICIPATION ACTIVITY 2.2.1: DrawingSim.

The task "Draw a square" can be decomposed into 8 steps. Use the interactive drawing tool to draw a square by following the given steps.

1. Put the pencil down on the paper.	**5.** Turn the pencil left 90 degrees.
2. Move the pencil forward.	**6.** Move the pencil forward.
3. Turn the pencil left 90 degrees.	**7.** Turn the pencil left 90 degrees.
4. Move the pencil forward.	**8.** Move the pencil forward.

Now try to draw your initials (press "Clear" first) . Can you decompose the task of drawing your initials into the most basic sequence of steps?



Once a task has been decomposed into its basic steps, then writing a program to solve that task becomes easier. A computer programmer often develops an ***algorithm***, which like a recipe, is a methodical step-by-step procedure to perform a task. Consider the basic steps for drawing a 3x3 tic-tac-toe grid:

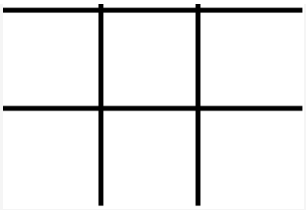


Figure 2.2.1: An algorithm for drawing a 3x3 tic-tac-toe grid.


```
N = 3 (The number of grid squares per side)

DRAW a vertical line with a length of N

MOVE the pencil to the right.

DRAW a vertical line with a length of N.

MOVE the pencil down and to the right.

DRAW a horizontal line with a length of N.

MOVE the pencil down.

DRAW a horizontal line with a length of N.
```

Now what if the programmer needs to draw a 4x4 grid, or a 10x10 grid? The process is the same but the number and length of the lines has changed. The programmer can write an algorithm that describes how to draw any N-sized grid:

Figure 2.2.2: An algorithm for drawing a NxN tic-tac-toe grid.

```
N is the number of grid squares per side

# Draw all the vertical lines
Do the following N - 1 times:
    DRAW a vertical line with a length of N
    MOVE the pencil to the right

# Draw all the horizontal lines
Do the following N - 1 times:
    MOVE the pencil down
    DRAW a horizontal line of length N
```


A programmer could then write a program using a programming language, such as Python, that follows the algorithm "recipe". Executing that program performs the given task, like creating an NxN

grid.

This method of evaluating a problem's most basic parts and then creating an algorithm to solve that problem is commonly known as ***computational thinking***.

PARTICIPATION ACTIVITY 2.2.2: Computational thinking concepts.

1) Task decomposition is the process of breaking a complex task into smaller parts.
   - ○ True
   - ○ False
2) An algorithm specifies a step-by-step procedure to perform a task.
   - ○ True
   - ○ False

2.3 The Python interactive interpreter

The ***Python interpreter*** is an application that can be used on various operating systems, including Microsoft Windows, Linux, and MacOS. 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 below animation illustrates how the Python interactive interpreter can be used as a calculator.

PARTICIPATION ACTIVITY 2.3.1: 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.

The interactive interpreter displays a ***prompt*** (">>>") that indicates the interpreter is ready to accept code. The user types a line of valid 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 above 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 above code are described elsewhere in the chapter.

The Python interpreter remembers previously-entered lines. A programmer can use the UP and DOWN arrow keys on the keyboard to review the executed commands, and optionally execute them again by pressing the enter key.

PARTICIPATION ACTIVITY 2.3.2: Match the Python terms with their definitions.

Code **Arrow keys** **Prompt** **Interpreter** **Line**

	A program that executes computer code.
	Scrolls through lines previously executed with the interpreter.
	The text that represents a computer program.
	Informs the programmer that the interpreter is ready to accept commands.
	A row of text.

Reset

2.4 Programming in Python

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.

PARTICIPATION ACTIVITY 2.4.1: Executing a simple Python program.

Animation captions:

1. The python interpreter reads a file line by line. Variables wage, hours, and salary are named references that refer to values stored by the interpreter.
2. 20 * 40 * 50 is computed, and then assigned to the variable salary.
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.

Important features to note about Python and the above code include:

- A ***statement*** is a program instruction. A program mostly consists of a series of statements, 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. * is the symbol 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.
- ***print()*** displays variables or expression values.
- '#' characters denote ***comments***, which are optional but can be used to explain portions of code to a human reader.
- Each part of the program is described in later sections

PARTICIPATION ACTIVITY 2.4.2: Executing Python code using the interpreter.

Click the run button below to execute the code on the left. Replace the values to calculate different salary amounts. Try changing the wage to 8 (U.S. minimum wage) or hours to 20 (half-time).

Load default template... **Run**

```
1 wage = 20   # Create a variable 'wage' w
2 hours = 40  # Create a variable 'hours'
3 weeks = 50  # Create a variable 'weeks'
4
5 salary = wage * hours * weeks  # Compute
6 print('Salary is:', end=' ')   # Print "
7 print(salary)                  # Print s
8
```

PARTICIPATION ACTIVITY 2.4.3: Race car game.

The below program simulates a race between two cars, displaying the position of each car at the end of the race. Make sure the output box below the code is visible, then run the program and observe the race.

The car1_top_speed and car1_acceleration variables control the maximum velocity and acceleration of car 1. Modify these variables, and run the program again. Can you make the second car win?.

You do not need to understand how the code works right now. Instead, just modify the speed and acceleration variables and observe how the output changes.

Load default template...

```
# Welcome to the Python 500 race! Click the run button to begin.

# Configurable values.
# Try changing car speeds, accelerations, and the simulation speed.
car1_top_speed = 60
car2_top_speed = 50

car1_acceleration = 11
car2_acceleration = 10

car1 = ['  ______\n',
        ' |__1_|_\\\n',
        '  O----O\n']


car2 = ['    ___\n',
        '   /__2_\\__\n',
```


Run

Many code editors color certain words, as in the above program, to assist a human reader understand various words' roles.

PARTICIPATION ACTIVITY 2.4.4: 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

2.5 Basic output

Printing of 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()***. Printing text is performed via: `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 "#".

Multiple print statements will each print on a new output line. However, using `print('Hello', end=' ')`, specifying *end=' '*, keeps the next print's output on the same line separated by a single space.

Figure 2.5.1: Printing text and new lines.

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

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



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

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

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 (such as `print("Don't eat that!")`).

PARTICIPATION ACTIVITY 2.5.1: Basic text output.

1) Which statement prints: *Welcome!*?

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

2) 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=' ')

PARTICIPATION ACTIVITY 2.5.2: Basic text output.

1) Type a statement that prints: *Hello*

Check Show answer

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

Figure 2.5.2: 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 2.5.3: Basic variable output.

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

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

PARTICIPATION ACTIVITY 2.5.4: Basic variable output.

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

Check Show answer

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. The programmer simply separates the items with commas; each item in the output will be separated by a space. Such combining can improve program readability, because the program's code corresponds more closely to the program's printed output.

Figure 2.5.3: Printing multiple items using a single print statement.

```
Wage: 20
Goodbye.
```

```
wage = 20

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

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

PARTICIPATION ACTIVITY 2.5.5: Single print statement.

Assume variable age = 22.

1) What is the output of

```
print('You are', age, 'years old.')
```

Check Show answer

A new line can also be output by inserting \n, known as a ***newline character***, within a string literal. For example, printing "1\n2\n3" prints each number on its own output line. \n consists of two characters, \ and n, but together are considered by the Python interpreter as a single newline character.

PARTICIPATION ACTIVITY 2.5.6: Output simulator.

The tool below allows for experimenting with print statements. 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.
```

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



```
Change this string!
```

CHALLENGE ACTIVITY 2.5.1: Generate output for given prompt.

(Sample solutions)

120640.2253466.qx3zqy7

Start

Complete the statement to produce the following output.
Note: Each space is underlined for clarity; you should output a space, not an underline.


```
p_q_r
```

```
print(                    )
```

1 2 3 4

Check Next

CHALLENGE ACTIVITY 2.5.2: Output simple text.

Write the simplest statement that prints the following on a single line.

```
3 2 1 Go!
```

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

120640.2253466.qx3zqy7

```
1
2 ''' Your solution goes here '''
3
```


Run

Run

CHALLENGE ACTIVITY 2.5.3: Output simple text with newlines.

Write a *single* statement that prints the following. Use newline (\n) characters.


```
3
2
1
Go!
```

Hint: print('3\n...')
Note: Whitespace (blank spaces / blank lines) matters; make sure your whitespace *exactly* matches the expected output.

120640.2253466.qx3zqy7

```
1
2 ''' Your solution goes here '''
3
```

Run

CHALLENGE ACTIVITY 2.5.4: 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.

120640.2253466.qx3zqy7

```
1
2 ''' Your solution goes here '''
3
```

Run

2.6 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 assign the entered text to the best_friend variable. input() causes the interpreter to wait until the user has entered some text and has pushed the return key.

Figure 2.6.1: Reading text input from a user.

```
print('Enter name of best friend:', end=' ')
best_friend = input()
print('My best friend is', best_friend)
```

```
Enter name of best friend: Marty McFly
My best friend is Marty McFly
```

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***, and is always surrounded by single or double quotes, for example 'Hello' or "#Goodbye# Amigo!".

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 2.6.1: 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 2.6.2: Reading user input.

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

Check Show answer

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.

However, often a programmer wants to read in an integer from the user, 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 2.6.2: 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 2.6.3: Converting user input to integers.

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

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

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

PARTICIPATION ACTIVITY 2.6.3: Converting user input to integers.

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

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

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

Figure 2.6.4: Basic input example.

```
hours = 40
weeks = 50
hourly_wage = int(input('Enter hourly wage: '))

print('\nSalary is', hourly_wage * hours * weeks)
```

```
Enter hourly wage: 12
Salary is 24000
...
Enter hourly wage: 20
Salary is 40000
```


PARTICIPATION ACTIVITY 2.6.4: Basic input.

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

Load default template...

```
1 dog_years = int(input('Enter age of dog
2 human_years = 7 * dog_years
3
4 print(dog_years, 'dog years is about', e
5 print (human_years, 'human years.')
6
```

3

Run

CHALLENGE ACTIVITY 2.6.1: Read user input and print to output.

Assign user_str with a string from user input, with the prompt: 'Enter a string: '
Hint -- Replace the ? in the following code:

```
user_str = ?('Enter a string: ')
```

Note: These activities may test code with different *pre-entered* test values, thus blank print statements are placed after the prompt so that output will appear on the following line. This activity will perform two tests: the first with user input of "Hello!", the second with user input of "3230 Main St.". See How to Use zyBooks.

120640.2253466.qx3zqy7

```
1
2 ''' Your solution goes here '''
3
4 print()
5
6 print(user_str)
```

Run

CHALLENGE ACTIVITY 2.6.2: Read user input numbers and perform a calculation.

Read two numbers from user input. Then, print the sum of those numbers.
Hint -- Complete the following code:

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


Note: These activities may test code with different test values. This activity will perform two tests: the first with num1 = 5 and num2 = 10, the second with num1 = 6 and num2 = 3. See How to Use zyBooks.

120640.2253466.qx3zqy7

```
1
```

```
2 ''' Your solution goes here '''
3
```


Run

2.7 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.

Figure 2.7.1: 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
```


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. Above, 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, saving much headache later.

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

PARTICIPATION ACTIVITY 2.7.1: 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 2.7.2: Common syntax errors.

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)

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

```
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

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

print(triangle_base)
print(*, end=' ')

print(triangle_height, end=' ')

print(') / 2 = ', end=' ')
print(triangle_area, end=' ')
print('cm^2')
```


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 2.7.3: 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.

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 2.7.2: 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. Common error types are listed below.

Table 2.7.1: Common error types.

Error type	Description
SyntaxError	The program contains invalid code that can not 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 2.7.4: 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.

ValueError | TypeError | SyntaxError | NameError | IndentationError

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

Reset

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 2.7.3: 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 to 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.

PARTICIPATION ACTIVITY 2.7.5: Fix the bug.

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

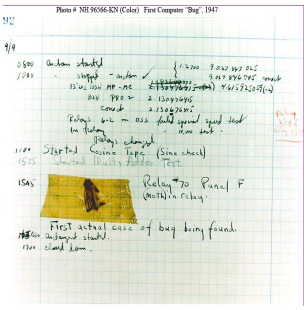
Load default template...

Run


```
1 num_beans = 500
2 num_jars = 3
3 total_beans = 0
4
5 print(num_beans, 'beans in', end=' ')
6 print(num_jars, 'jars yields', end=' ')
7 total_beans = num_beans * num_jars
8 print('total_beans', 'total')
9
```

Figure 2.7.4: The first bug.

As a side-note, the term "bug" to describe a runtime error was popularized when in 1947 engineers 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, still preserved today (http://en.wikipedia.org/wiki/Computer_bug).



CHALLENGE ACTIVITY 2.7.1: 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)
```


120640.2253466.qx3zqy7

```
1
2 ''' Your solution goes here '''
3
```

Run

2.8 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.

The following program outputs a simple triangle.

Figure 2.8.1: Output art: Printing a triangle.

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

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

PARTICIPATION ACTIVITY 2.8.1: Create ASCII art.

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

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

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

Load default template...

Run

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

PARTICIPATION ACTIVITY 2.8.2: Create ASCII art.

Complete the cat drawing below. Note the '\' character is actually displayed by printing the two character sequence '\\'.


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

Load default template...

Run

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

```
3
```


PARTICIPATION ACTIVITY 2.8.3: Create ASCII art.

Be creative: Print something you'd like to see, more impressive than above.

No template provided

Run

```
1
```

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 (someitem)" 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 above.

2.9 Development environment

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 http://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 http://www.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 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.org) 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.

PARTICIPATION ACTIVITY 2.9.1: 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

2.10 Computers and programs

Figure 2.10.1: Looking under the hood of a car.

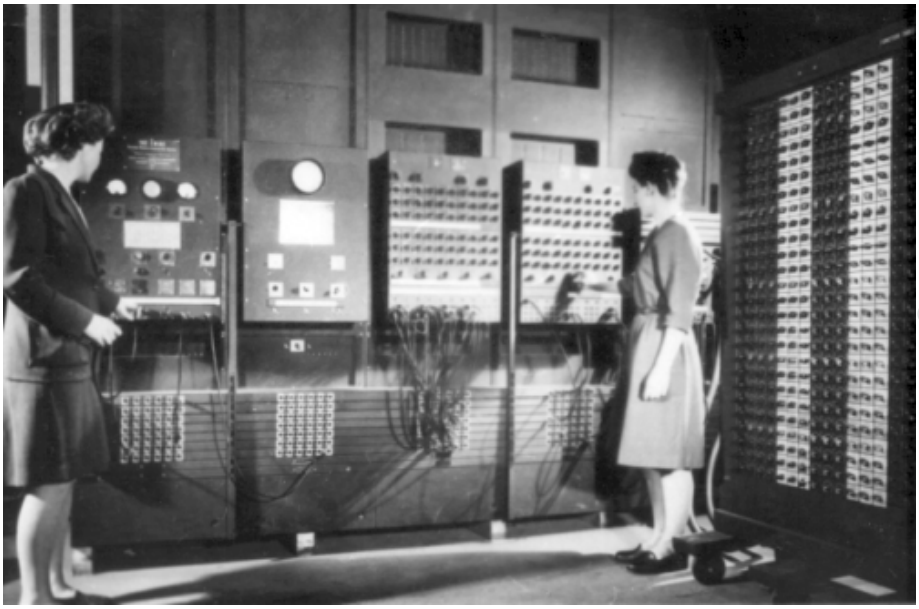
Source: Robert Couse-Baker / CC-BY-2.0 via Wikimedia Commons (Original image cropped)

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 sections provides a very brief introduction.

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

Figure 2.10.2: Early computer made from thousands of switches.

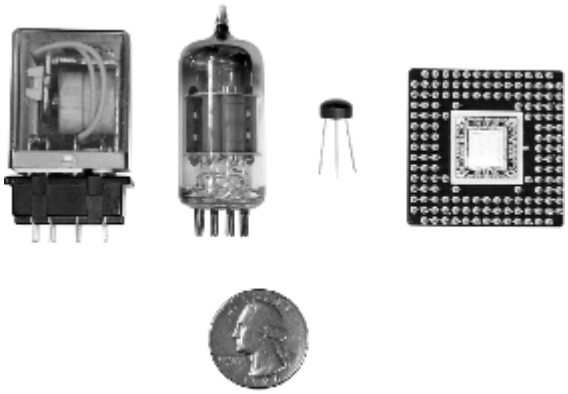


Source: ENIAC computer (U. S. Army Photo / Public domain)

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.

To support different calculations, circuits called ***processors*** were created to process (aka *execute*) a list of desired calculations, each calculation called ***instruction***. The instructions were specified by configuring external switches, as in the figure on the left. 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 2.10.3: 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 2.10.4: Memory.

Thus, a computer is basically a processor interacting with a memory, as depicted in the following example. In the 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 50 (weeks/year), outputting the final result to the screen.

PARTICIPATION ACTIVITY 2.10.2: Computer processor and memory.

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 written to the screen.

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

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 2.10.1: 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 2.10.3: Memory stores instructions and data as 0s and 1s.

Animation captions:

1. Memory stores instructions and data as 0s and 1s.
2. The material 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 2.10.4: 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 0 is assigned to 'Next'.
4. The processor executes the instruction at location 0, performing the same sequence of instructions over and over again.

PARTICIPATION ACTIVITY 2.10.5: 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) A memory stores bits.
   - ○ True
   - ○ False

4) The computer inside a modern smartphone would have been huge 30 years ago.
   - ○ True
   - ○ False

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

- True
- False

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 1950s 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 Translation") or *ALGOL* (for "Algorithmic Language") languages, 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 2.10.6: Program compilation and execution.

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 2.10.7: Programs.

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

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; an assembler translates to machine instructions (0s and 1s).

Reset

Note to 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.

2.11 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 smart phones. The term computer even refers to computing devices embedded inside other electronic devices such as medical equipment, automobiles, aircraft, consumer electronics, military systems, etc.

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):

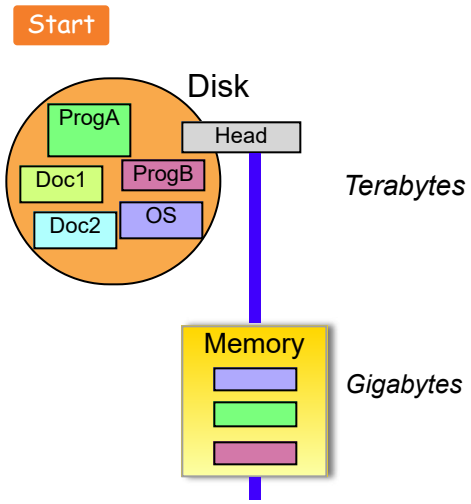
- ***Input/output devices***: A ***screen*** (or monitor) displays items to a user. The above examples displayed textual items, but today's computers display graphical items too. A ***keyboard*** allows a

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 which 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, a photo editing program, etc. 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 2.11.1: Some computer components.



View animation caption(s)

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.[ML] 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 2.11.2: Programs.

Moore's Law **Disk** **Clock** **Cache** **Operating system** **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.

	Rate at which a processor executes instructions.
	The doubling of IC capacity roughly every 18 months.

Reset

A side-note: 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.

(*ML) 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.

2.12 Language history

In the 1940s, programmers originally wrote each instruction using 0s and 1s, such as "001 1100001 001001 1100010", eventually creating enough instructions that a new "language" emerged known as ***machine language***. Machine language is founded on a two-character alphabet. The two characters correspond to two states, either a 1 or a 0, or equivalently a true or a false state. Particular sequences of 0s and 1s represent specific ***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 an executable program (thus using the assembler program to help programmers create more complex programs).

In the 1950s and 1960s, programmers created ***high-level languages*** to support programming using formulas or algorithms, such as the *FORTRAN* (for "Formula Translator") or *ALGOL* (for "Algorithmic

Language") languages, consisting of higher-level instructions known as ***statements*** such as `F = (9 * C) / 5 + 32`. Such statements are more closely related to how humans think than are machine instructions or assembly instructions, helping programmers create more complex programs. Programmers also created ***compilers*** to automatically translate those high-level language programs, even higher-level than assembly, into assembly-language programs.

PARTICIPATION ACTIVITY 2.12.1: Programming evolution.

1) An assembler translates an assembly program into machine instructions.
   - True
   - False

2) A high-level language eases a programmer's task of writing complex programs.
   - True
   - False

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 3.0 was released in 2008 to rectify various language design issues. However, Python 2.7 is the most widely used version, 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 are becoming more widely used as new projects adopt 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 December 2015 survey that measured the popularity of various programming languages found that Python (4.4%) is the second most popular language, just behind C++ (5.9%). (Source: www.tiobe.com). A review of open-source project contributions from 2004 to 2013 shows that the ratio of contributions that are Python more than doubled, while C/C++ contributions fell 5-10% and Java contributions remained the same. (Source: www.ohloh.net).

PARTICIPATION ACTIVITY 2.12.2: 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

©zyBooks 02/01/22 07:06 1126733
Hector Chicas Castillo
WGUC9492018