1.1 Computers and programs (general)

Figure 1.1.1: Looking under the hood of a car.

Source: zyBooks

Feedback?

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.

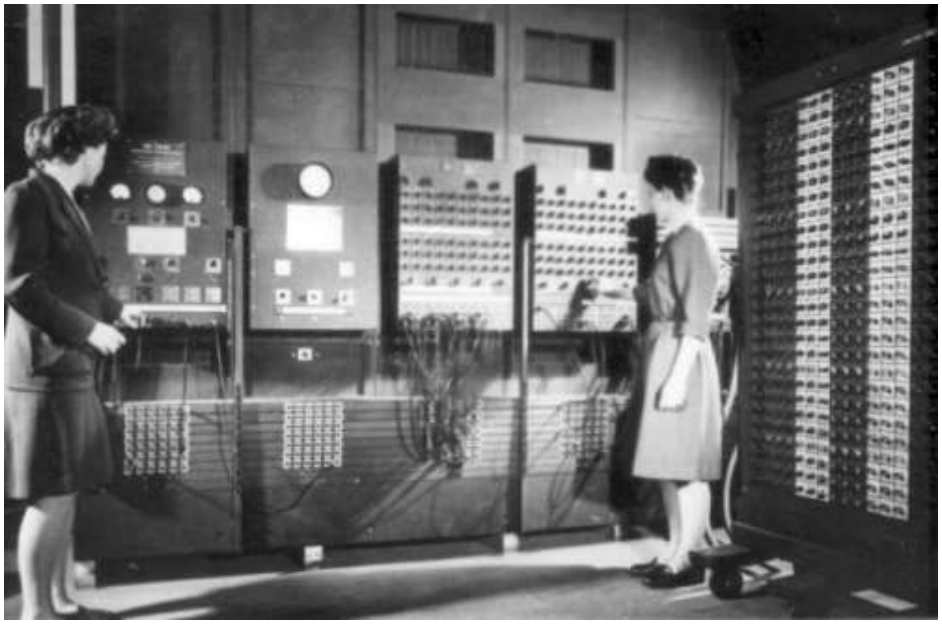
Switches

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

Feedback?

Figure 1.1.2: Early computer made from thousands of switches.



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

Feedback?

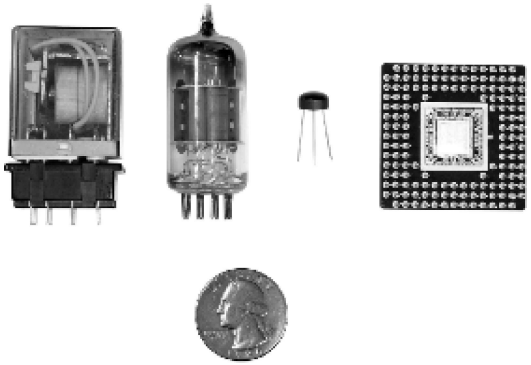
These circuits became increasingly complex, leading to the first electronic computers in the 1930s and 1940s, consisting of about ten thousand electronic switches and typically occupying entire rooms as in the above figure. Early computers performed thousands of calculations per second, such as calculating 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 1.1.3: As switches shrunk, so did computers. The computer processor chip on the right has millions of switches.



Source: zyBooks

Feedback?

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.

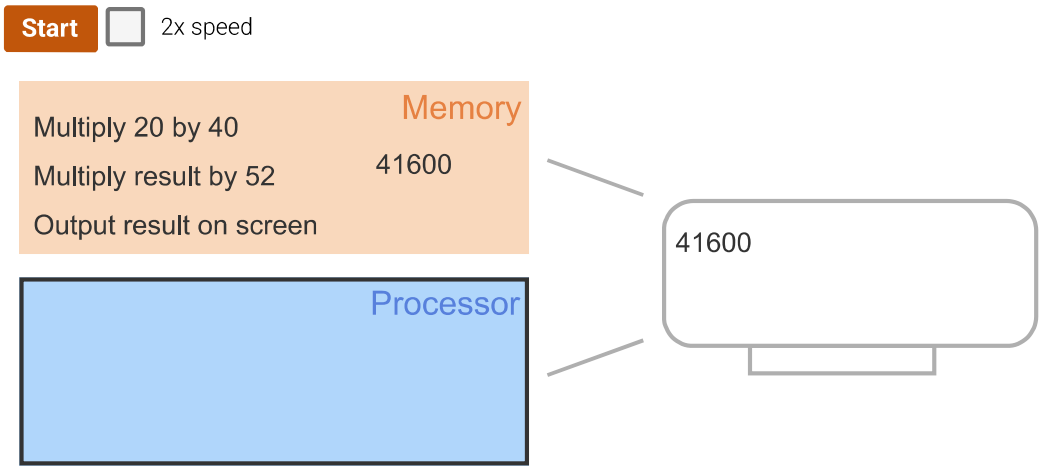
Figure 1.1.4: Memory.

Feedback?

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 | 1.1.2: Computer processor and memory.



Captions

Feedback?

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 1.1.1: 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*

Feedback?

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 1.1.3: Memory stores instructions and data as 0s and 1s.

Start 2x speed

Location	Memory	Meaning
0	011 1100001 001001 1100010	Mul 97, #9, 98
1	100 1100010 000101 1100010	Div 98, #5, 98
2	001 1100010 100000 1100011	Add 98, #32, 99
3	101 0000000000000000000	Jmp 0
4	??	
...		
96	??	
97	00000000000000000010100	20
98	??	
99	??	

Location	Memory
0	Mul 97, #9, 98
1	Div 98, #5, 98
2	Add 98, #32, 99
3	Jmp 0
4	??
96	??
97	20
98	??
99	??

Captions

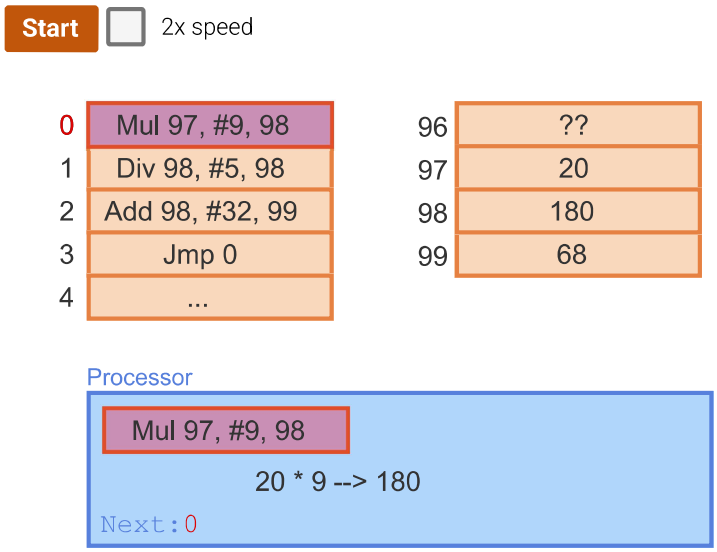
Feedback?

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 1.1.4: Processor executing instructions.



Captions

Feedback?

PARTICIPATION ACTIVITY 1.1.5: 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 such as Add 200, #9, 201, represented as 0s and 1s.

- ○ True
- ○ False

Feedback?

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 1.1.6: Program compilation and execution.

Start 2x speed

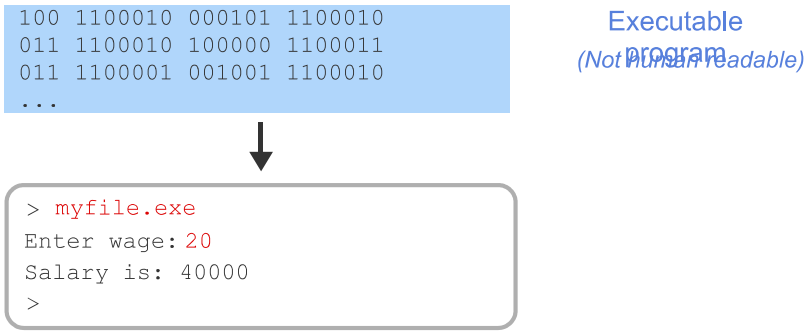
myfile.txt

```
put "Enter wage: "
hourlyWage = Get next input
put "Salary is: "
put (hourlyWage * 40 * 50)
```

High level program

myfile.exe

```
...
011 1100001 001001 1100010
```



Captions

Feedback?

PARTICIPATION ACTIVITY | 1.1.7: Programs.

How to use this tool

Assembly language **Machine instruction** **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

Feedback?

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.

1.2 Programming (general)

Computer program basics

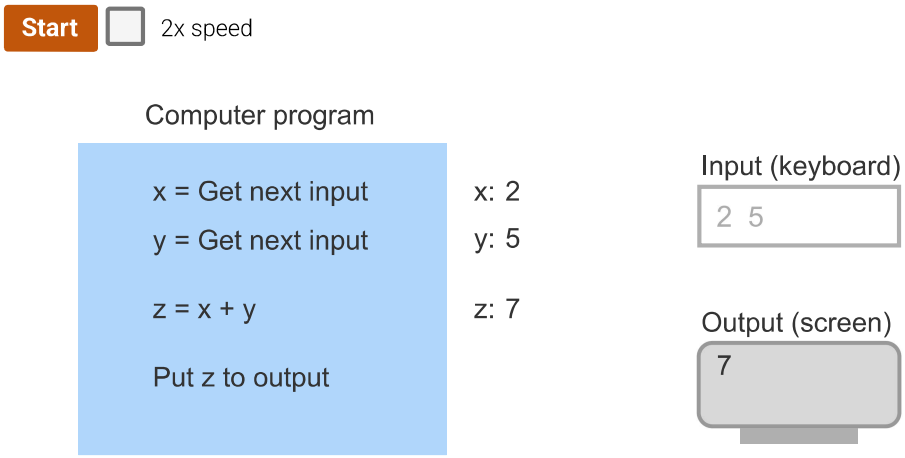
Computer programs are abundant in many people's lives today, carrying out applications on smartphones, tablets, and laptops, powering businesses like Amazon and Netflix, helping cars drive and planes fly, and much more.

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 1.2.1: A basic computer program.



Captions

Feedback?

PARTICIPATION ACTIVITY 1.2.2: A basic computer program.

Consider the example above.

1) The program has ____ instructions.

Check Show answer

2) Suppose a new instruction were inserted as follows:

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

What would the last instruction then 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

Feedback?

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 1.2.3: 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 (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

put m

m = m * 2
put m

m = m * m
put m

m = m + 15
put m
```

m:

Feedback?

PARTICIPATION ACTIVITY 1.2.4: 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

Feedback?

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 | 1.2.5: 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).

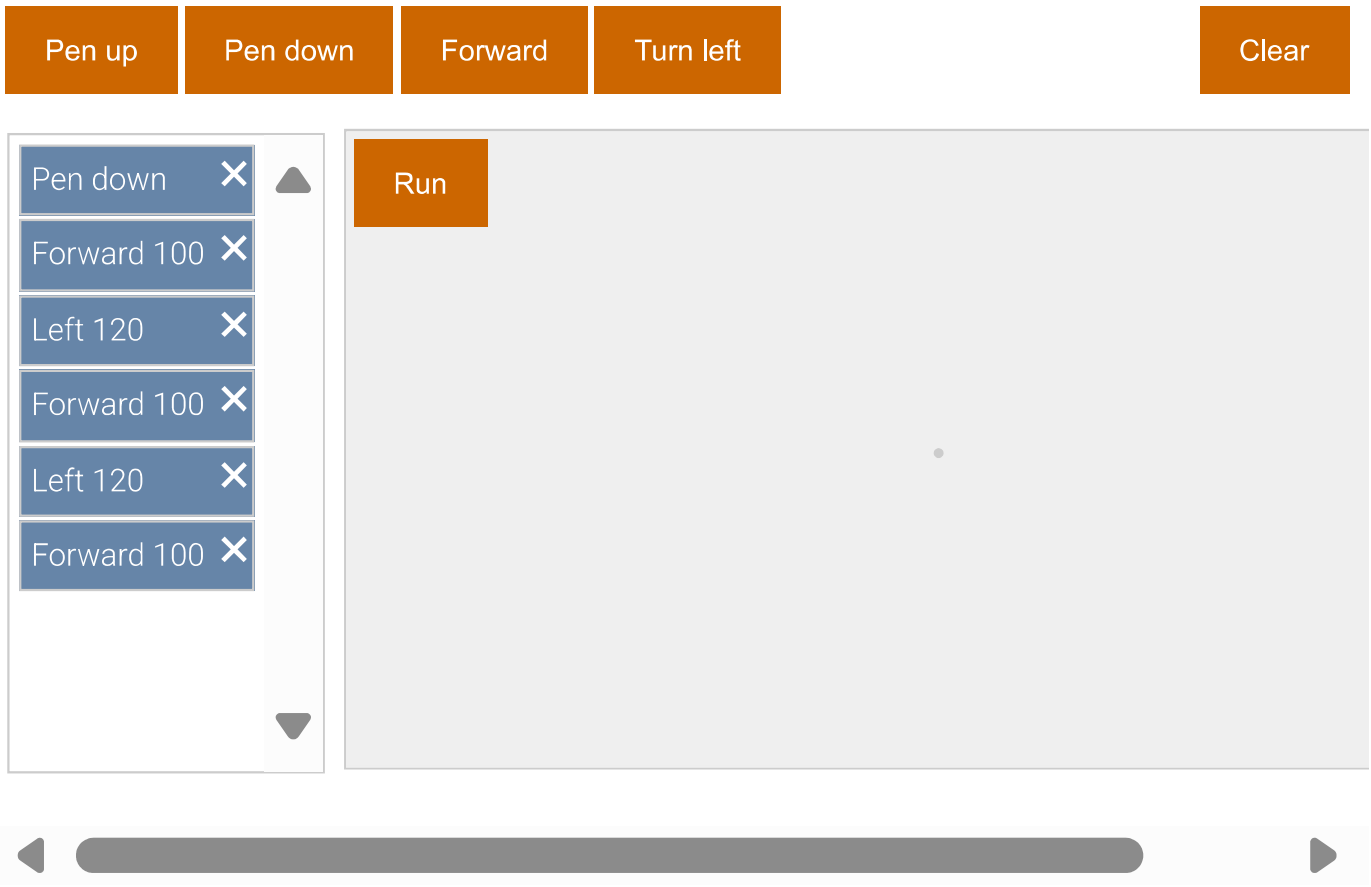
The 6-instruction algorithm shown below ("Pen down", "Forward 100", etc.) draws a triangle.

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.

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.



Feedback?

1.3 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 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. In fact, many libraries now support Python 3.x since Python 2.7 has an "End Of Life" date (no more bug fixes) set for 2020 (Source: Python.org). 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 1.3.1: 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

1.4 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 1.4.1: The Python interpreter.

Start 2x speed

```
>>> wage = 20
>>> hours = 40
>>> weeks = 50
>>> salary = wage * hours * weeks
>>> print(salary)
40000
>>> hours = 35
>>> salary = wage * hours * weeks
>>> print(salary)
35000
>>>
```

Python interpreter

Name	Value
wage	20
hours	35
weeks	50
salary	35000

Captions

Feedback?

PARTICIPATION ACTIVITY 1.4.2: Match the Python terms with their definitions.

How to use this tool

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

Feedback?

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 | 1.4.3: Executing a simple Python program.

Start ☐ 2x speed

file.py

```
wage = 20
hours = 40
weeks = 50
salary = wage * hours * weeks

print('Salary is:', salary)

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

Python interpreter

```
>>>
```

Name	Value
wage	20
hours	35
weeks	50
salary	35000

```
Salary is: 40000
New salary is: 35000
```

Captions

Feedback?

PARTICIPATION ACTIVITY | 1.4.4: 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

Feedback?

zyDE 1.4.1: A first program.

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 click "run."

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...

```
1  # Welcome to the Python 500 race! Click the run button to
2
3  # Configurable values.
4  # Try changing car speeds, accelerations, and the simulati
5  car1_top_speed = 60
6  car2_top_speed = 50
7
8  car1_acceleration = 11
9  car2_acceleration = 10
10
11 car1 = ['   ______\n',
12         ' |__1_|_\\\n',
13         '  O----O\n']
14
15
16 car2 = ['     ___\n',
17         '    /__2_\\__\n',
18         '     O----O\n']
```

Run

Feedback?

1.5 Basic input and output

Basic text 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 #.

Each print statement will output on a new line. A new output line starts after each print statement, called a ***newline***. A print statement's default behavior is to automatically end with a newline. However, using `print('Hello', end=' ')`, specifying *end=' '*, keeps the next print's output on the same line separated by a single space. Any space, tab, or newline is called ***whitespace***.

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!")`*).*

Figure 1.5.1: 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?
```

Feedback?

PARTICIPATION ACTIVITY 1.5.1: Basic text output.

1) Select the statement that prints the following: *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=' ')

Feedback?

PARTICIPATION ACTIVITY 1.5.2: Basic text output.

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

Check Show answer

Feedback?

CHALLENGE ACTIVITY 1.5.1: 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.

Learn how our autograder works

679280.5289994.qx3zqy7

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

Run

Feedback?

CHALLENGE ACTIVITY 1.5.2: 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.

Learn how our autograder works

679280.5289994.qx3zqy7

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

Run

Feedback?

Outputting a variable's value

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

Figure 1.5.2: Printing the value of a variable.

```
wage = 20

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

```
Wage is
20
Goodbye.
```

Feedback?

PARTICIPATION ACTIVITY 1.5.3: Basic variable output.

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

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

Feedback?

PARTICIPATION ACTIVITY 1.5.4: Basic variable output.

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

Check Show answer

Feedback?

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. 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 1.5.3: Printing multiple items using a single print statement.

```
wage = 20

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

```
Wage: 20
Goodbye.
```

Feedback?

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 1.5.4: Printing using newline characters.

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

```
1
2
3
```

Feedback?

Using print() by itself without any text also prints a single newline.

Figure 1.5.5: printing without text.

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

```
123

abc
```

Feedback?

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 1.5.5: 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!
```

Feedback?

PARTICIPATION ACTIVITY 1.5.6: 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

Feedback?

CHALLENGE ACTIVITY | 1.5.3: Enter the output.

Type the program's output.

Note:

- `print(string1, string2)` adds a single space between string1 and string2.
- `print()` automatically adds a newline after the output. So, **don't forget to press the *Enter* or *Return* key** to add an additional newline as needed.

Click here for example

679280.5289994.qx3zqy7

Start

Type the program's output

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

```
Joe is happy.
```

1 2 3

Check Next

Feedback?

Basic input

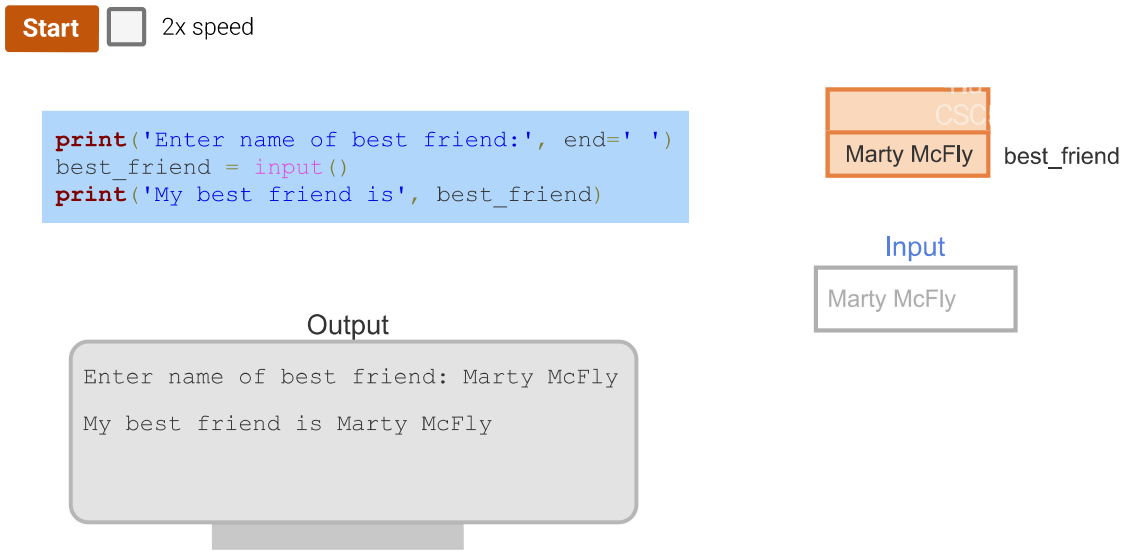
Many useful programs allow a user to enter values, such as typing a number, a name, etc.

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. 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*** 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 1.5.7: A program can get an input value from the keyboard.



Captions

Feedback?

PARTICIPATION ACTIVITY 1.5.8: Reading user input.

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

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

Feedback?

PARTICIPATION ACTIVITY 1.5.9: Reading user input.

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

Check Show answer

Feedback?

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 1.5.6: Using int() to convert strings to integers.

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

print(my_string)
print(my_int)
```

```
123
123
```

Feedback?

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 1.5.7: 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
```

Feedback?

PARTICIPATION ACTIVITY 1.5.10: 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

Feedback?

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 1.5.8: Basic input example.

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

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

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

Feedback?

zyDE 1.5.1: Basic input.

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

Load default template...

```
1  human_years = int(input('Enter age of dog (in human years):
2  print()
3
4  dog_years = 7 * human_years
5
6  print(human_years, 'human years is about', end=' ')
7  print(dog_years, 'dog years.')
8
9
10
```

3

Run

Feedback?

CHALLENGE ACTIVITY 1.5.4: 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)
```

See How to Use zyBooks for info on how our automated program grader works.

679280.5289994.qx3zqy7

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

Run

Feedback?

CHALLENGE ACTIVITY 1.5.5: Read user input and print to output.

The given code reads in two integers from input and assigns them to num1 and num2, respectively.

Complete the program to:

1. Read a third integer from input into variable **num3**.

2. Output the product of the three integers by placing `num1 * num2 * num3` in a print statement.

Click here for example 1

Click here for example 2

Learn how our autograder works

679280.5289994.qx3zqy7

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

""" Your solution goes here """

```

Run

Feedback?

1.6 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 1.6.1: 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
```

Feedback?

PARTICIPATION ACTIVITY 1.6.1: 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

Feedback?

PARTICIPATION ACTIVITY 1.6.2: Common syntax errors.

Find and click on the 3 syntax errors.

1)

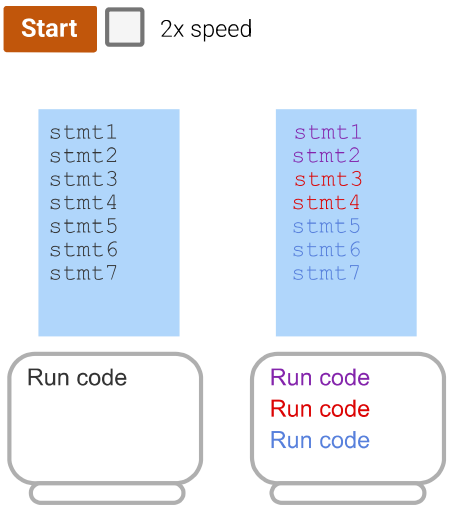
```
triangle_base = 0    # Triangle base (cm)
triangle_height = 0  # Triangle height (cm)
triangle_area = 0
      # Triangle area (cm)

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')
```

Feedback?

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 1.6.3: Run code frequently to avoid many errors.



Captions

Feedback?

PARTICIPATION ACTIVITY 1.6.4: Testing for syntax errors.

1) Experienced programmers write an entire program before running and testing the code.
   - ○ True
   - ○ False

Feedback?

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 1.6.2: 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'
```

Feedback?

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 1.6.1: 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.

Feedback?

PARTICIPATION ACTIVITY 1.6.5: 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.

How to use this tool

ValueError | IndentationError | TypeError | SyntaxError | NameError

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

Reset

Feedback?

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 1.6.3: 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
```

Feedback?

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 1.6.1: Fix the bug.

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

Load default template...

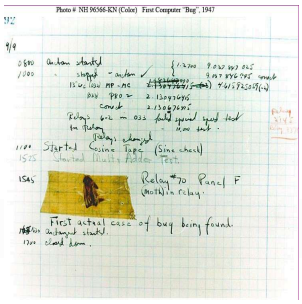
Run

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

Feedback?

Figure 1.6.4: 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).



Feedback?

CHALLENGE ACTIVITY 1.6.1: 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)
```

Learn how our autograder works

679280.5289994.qx3zqy7

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

Run

Feedback?

1.7 Development environment

IDEs

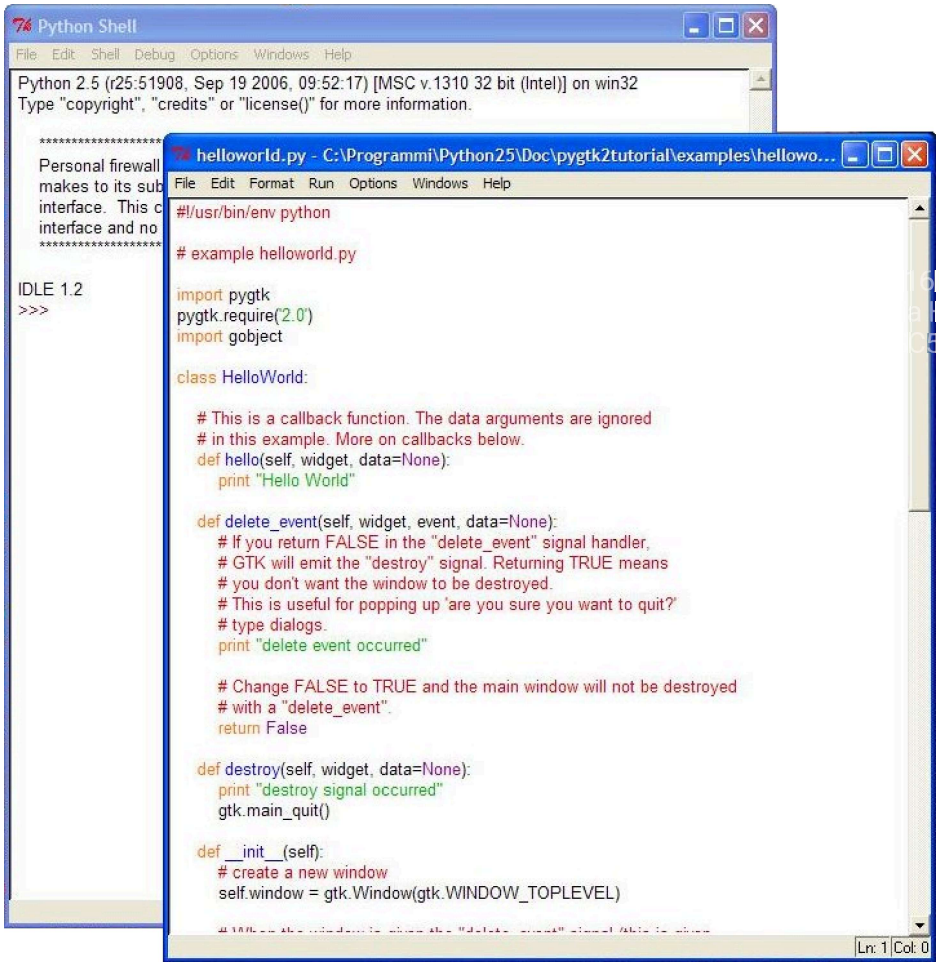
This web material embeds a Python interpreter so that the reader may experiment with Python programming. However, for normal development, a programmer installs Python as an application on a local computer. Macintosh and Linux operating systems usually include Python, while Windows does not. Programmers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

Figure 1.7.1: IDLE environment for coding and running Python.



Source: Katanzag via Wikimedia Commons (CC BY 3.0)

Feedback?

PARTICIPATION ACTIVITY 1.7.1: 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

Feedback?

1.8 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 1.8.1: Precisely formatting a meeting invite.

Start 2x speed

Kia Smith is inviting you to a video meeting.

Join meeting:
http://www.zoomskype.us/5592

Phone:
1-669-555-2634 (San Jose)
1-929-555-4000 (New York)

Meeting ID: 5592

Reminder: 10 min before

Kia Smith is inviting you to a video meeting. Join meeting:

http://www.zoomskype.us/5592 Phone:
1-669-555-2634 (San Jose)
1-929-555-4000 (New York)

Meeting ID: 5592
---------- Reminder: 10 min before

Captions

Feedback?

PARTICIPATION ACTIVITY 1.8.2: 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

Feedback?

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 1.8.3: 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 printIn
Printing A linE is done using print1n

Check Show answer

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

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, 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

Feedback?

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.

1.9 Python example: Salary calculation

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

zyDE 1.9.1: Executing Python code using the interpreter.

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

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

Load default template... Run

```
1  hourly_wage = 20
2
3  print('Annual salary is: ')
4  print(hourly_wage * 40 * 50)
5  print()
6
7  print('Monthly salary is: ')
8  print(((hourly_wage * 40 * 5
9  print()
10
11 # FIXME: The above is wrong.
12 #        the 1 so that the s
13 #        outputs monthly sal
14
15
16
```

Feedback?

1.10 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 1.10.1: Output art: Printing a triangle.

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

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

Feedback?

zyDE 1.10.1: 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

```
1 print('   *   ')
2 print('  ***  ')
3 print(' ***** ')
4 print('*******')
5
```

Feedback?

zyDE 1.10.2: Create ASCII art.

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

Load default template...

Run

```
1 print('/\\       /\\')
2 print('  o     ')
3
```

Feedback?

zyDE 1.10.3: 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
```

Feedback?

1.11 zyLab training: Basics

While the zyLab platform can be used without training, a bit of training may help some students avoid common issues.

The assignment is to get an integer from input, and output that integer squared, ending with newline. (Note: This assignment is configured to have students programming directly in the zyBook. Instructors may instead require students to upload a file). Below is a program that's been nearly completed for you.

1. Click "Run program". The output is wrong. Sometimes a program lacking input will produce wrong output (as in this case), or no output. Remember to always pre-enter needed input.
2. Type 2 in the input box, then click "Run program", and note the output is 4.
3. Type 3 in the input box instead, run, and note the output is 6.

When students are done developing their program, they can submit the program for automated grading.

1. Click the "Submit mode" tab
2. Click "Submit for grading".
3. The first test case failed (as did all test cases, but focus on the first test case first). The highlighted arrow symbol means an ending newline was expected but is missing from your program's output.

Matching output exactly, even whitespace, is often required. Change the program to output an ending newline.

1. Click on "Develop mode", and change the output statement to output a newline: `print(userNumSquared)`. Type 2 in the input box and run.
2. Click on "Submit mode", click "Submit for grading", and observe that now the first test case passes and 1 point was earned.

The last two test cases failed, due to a bug, yielding only 1 of 3 possible points. Fix that bug.

1. Click on "Develop mode", change the program to use * rather than +, and try running with input 2 (output is 4) and 3 (output is now 9, not 6 as before).
2. Click on "Submit mode" again, and click "Submit for grading". Observe that all test cases are passed, and you've earned 3 of 3 points.

679280.5289994.qx3zqy7

LAB ACTIVITY | 1.11.1: zyLab training: Basics | 0 / 3

main.py — Load default template...

```
1 userNum = int(input())
2 userNumSquared = userNum + userNum   # Bug here; fix it when instructed
3
4 print(userNumSquared, end=' ')       # Output formatting issue here; fix it when instru
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.py (Your program) → Output

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

Feedback?

1.12 zyLab training: Interleaved input / output

Auto-graded programming assignments have numerous advantages, but have some challenges too. Students commonly struggle with realizing that example input / output provided in an assignment's specification interleaves input and output, but the program *should only output the output parts*. If a program should double its input, an instructor might provide this example:

```
Enter x:
5
x doubled is: 10
```

Students often incorrectly create a program that outputs the 5. Instead, the program should only output the output parts:

```
Enter x:
x doubled is: 10
```

The instructor's example is showing both the output of the program, AND the user's input to that program, assuming the program is developed in an environment where a user is interacting with a program. But the program itself doesn't output the 5 (or the newline following the 5, which occurs when the user types 5 and presses enter).

Also, if the instructor configured the test cases to observe whitespace, then according to the above example, the program should output a newline after `Enter x:` (and possibly after the 10, if the instructor's test case expects that).

The program below *incorrectly* echoes the user's input to the output.

1. Try submitting it for grading (click "Submit mode", then "Submit for grading"). Notice that the test cases fail. The first test case's highlighting indicates that output 3 and newline were not expected. In the second test case, the -5 and newline were not expected.
2. Remove the code that echoes the user's input back to the output, and submit again. Now the test cases should all pass.

679280.5289994.qx3zqy7

LAB ACTIVITY | 1.12.1: zyLab training: Interleaved input / output | 0 / 2

main.py — Load default template...

```
1 print('Enter x: ')
2 x = int(input())
3
```

```
print(x) # Student mistakenly is echoing the input to output to match example
print('x doubled is:', (2 * x))
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → main.py (Your program) → Output

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

Feedback?

1.13 LAB: Formatted output: Hello World!

Write a program that outputs "Hello World!" For ALL labs, end with newline (unless otherwise stated).

679280.5289994.qx3zqy7

LAB ACTIVITY | 1.13.1: LAB: Formatted output: Hello World! | 0 / 10

main.py | Load default template...

```
1 ''' Type your code here. '''
2
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.py** (Your program) → Output

Program output displayed here

Coding trail of your work [What is this?](#)

History of your effort will appear here once you begin working on this zyLab.

Feedback?

1.14 LAB: Formatted output: No parking sign

Write a program that prints a formatted "No parking" sign as shown below. Note the first line has two leading spaces. For ALL labs, end with newline (unless otherwise stated).

```
  NO PARKING
2:00 - 6:00 a.m.
```

679280.5289994.qx3zqy7

LAB ACTIVITY | 1.14.1: LAB: Formatted output: No parking sign | 0 / 10

main.py — Load default template...

```
1 ''' Type your code here. '''
2
3
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.py** (Your program) → Output

Program output displayed here

Coding trail of your work [What is this?]

```
History of your effort will appear here once you begin working
on this zyLab.
```

Feedback?

1.15 LAB: Input: Welcome message

Write a program that takes a first name as the input, and outputs a welcome message to that name.

Ex: If the input is Pat, the output is:

```
Hello Pat and welcome to CS Online!
```

679280.5289994.qx3zqy7

LAB ACTIVITY | 1.15.1: LAB: Input: Welcome message | 0 / 10

main.py — Load default template...

```
1 user_name = input()
2
3 ''' Type your code here. '''
```

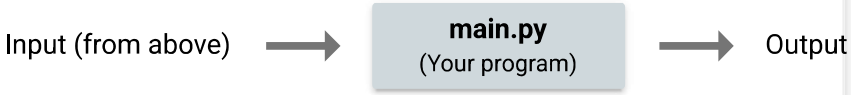
Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

Feedback?

1.16 LAB: Input: Mad Lib

Mad Libs are activities that have a person provide various words, which are then used to complete a short story in unexpected (and hopefully funny) ways.

Complete a program that reads four values from input and stores the values in variables `first_name`, `generic_location`, `whole_number`, and `plural_noun`. The program then uses the input values to output a short story. The first input statement is provided in the code as an example.

Notes: To test your program in the Develop mode, pre-enter four values (in separate lines) in the input box and click the Run program button. The auto-grader in the Submit mode will test your program with different sets input of values.

Ex: If the input values are:

```
Eric
Chipotle
```

```
12
cars
```

then the program uses the input values and outputs a story:

```
Eric went to Chipotle to buy 12 different types of cars
```

Ex: If the input values are:


```
Brenda
Philadelphia
6
bells
```

then the program uses the input values and outputs a story:

```
Brenda went to Philadelphia to buy 6 different types of bells
```

679280.5289994.qx3zqy7

LAB ACTIVITY | 1.16.1: LAB: Input: Mad Lib | 0 / 10

main.py — Load default template...

```
# Read a value from a user and store the value in first_name
first_name = input()

# TODO: Type your code to read three more values here.


# Output a short story using the four input values. Do not modify the code below.
print(first_name, 'went to', generic_location, 'to buy', whole_number, 'different types
```

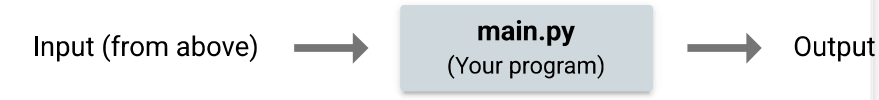
Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

Feedback?

1.17 LAB: Warm up: Hello world

This zyLab activity prepares a student for a full programming assignment. Warm up exercises are typically simpler and worth fewer points than a full programming assignment, and are well-suited for an in-person scheduled lab meeting or as self-practice.

For each of the following steps, end the program's output with a newline.

(1) Write a program that outputs the following. (Submit for 1 point).

```
Hello world!
```

(2) Update to output the following. (Submit again for 1 more point, so 2 points total).

```
Hello world!
How are you?
```

(3) Finally, update to output the following. (Submit again for 1 more point, so 3 points total).

```
Hello world!
How are you?
   (I'm fine).
```

Hint: The ' character is printed by the two character sequence \'. Ex: print('\'')

679280.5289994.qx3zqy7

LAB ACTIVITY | 1.17.1: LAB: Warm up: Hello world | 0 / 3

main.py — Load default template...

```
1 # Type your code here
```

Develop mode | **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.py** (Your program) → Output

Program output displayed here

Coding trail of your work [What is this?]

```
History of your effort will appear here once you begin working
on this zyLab.
```

Feedback?

1.18 LAB: Warm up: Basic output with variables

This zyLab activity prepares a student for a full programming assignment. Warm up exercises are typically simpler and worth fewer points than a full programming assignment, and are well-suited for an in-person scheduled lab meeting or as self-practice.

A variable like user_num can store a value like an integer. Extend the given program as indicated.

1. Output the user's input. (2 pts)
2. Output the input squared and cubed. *Hint: Compute squared as user_num * user_num.* (2 pts)
3. Get a second user input into user_num2, and output the sum and product. (1 pt)

Note: This zyLab outputs a newline after each user-input prompt. For convenience in the examples below, the user's input value is shown on the next line, but such values don't actually appear as output when the program runs.

```
Enter integer:
4
You entered: 4
4 squared is 16
And 4 cubed is 64 !!
Enter another integer:
5
4 + 5 is 9
4 * 5 is 20
```


679280.5289994.qx3zqy7

LAB ACTIVITY 1.18.1: LAB: Warm up: Basic output with variables 0 / 5

main.py Load default template...

```
1 user_num = int(input('Enter integer:\n'))
```

```
2
3 # Type your code here
```

Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.py** (Your program) → Output

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

Feedback?

1.19 LAB*: Program: ASCII art

This zyLab activity is the traditional programming assignment, typically requiring a few hours over a week. The previous sections provide warm up exercises intended to help a student prepare for this programming