Chapter 1 Notes.

* The ***Central Processing Unit* (or CPU)** is the part of the computer that is

built to be obsessed with “what is next?” If your computer is rated at 3.0

Gigahertz, it means that the CPU will ask “What next?” three billion times

per second. You are going to have to learn how to talk fast to keep up with

the CPU.

* The ***Main Memory***is used to store information that the CPU needs in a

hurry. The main memory is nearly as fast as the CPU. But the information

stored in the main memory vanishes when the computer is turned off.

* The ***Secondary Memory***is also used to store information, but it is much

slower than the main memory. The advantage of the secondary memory is

that it can store information even when there is no power to the computer.

Examples of secondary memory are disk drives or flash memory (typically

found in USB sticks and portable music players).

* The ***Input and Output Devices***are simply our screen, keyboard, mouse, microphone,

speaker, touchpad, etc. They are all of the ways we interact with

the computer.

* These days, most computers also have a *Network Connection* to retrieve

information over a network. We can think of the network as a very slow

place to store and retrieve data that might not always be “up”. So in a sense,

the network is a slower and at times unreliable form of *Secondary Memory*.

* First, you need to know the programming language (Python) - you need to

know the vocabulary and the grammar. You need to be able to spell the

words in this new language properly and know how to construct well-formed

“sentences” in this new language.

* Second, you need to “tell a story”. In writing a story, you combine words

and sentences to convey an idea to the reader. There is a skill and art in

constructing the story, and skill in story writing is improved by doing some

writing and getting some feedback. In programming, our program is the

“story” and the problem you are trying to solve is the “idea”.

The reserved words in the language where humans talk to Python include the

following:

A screenshot of a cell phone

Description automatically generated

**input** Get data from the “outside world”. This might be reading data from a

file, or even some kind of sensor like a microphone or GPS. In our initial

programs, our input will come from the user typing data on the keyboard.

**output** Display the results of the program on a screen or store them in a file or

perhaps write them to a device like a speaker to play music or speak text.

**sequential execution** Perform statements one after another in the order they

are encountered in the script.

**conditional execution** Check for certain conditions and then execute or skip a

sequence of statements.

**repeated execution** Perform some set of statements repeatedly, usually with

some variation.

**reuse** Write a set of instructions once and give them a name and then reuse those

instructions as needed throughout your program.

**Syntax errors** These are the first errors you will make and the easiest to fix. A

syntax error means that you have violated the “grammar” rules of Python.

Python does its best to point right at the line and character where it noticed

it was confused. The only tricky bit of syntax errors is that sometimes the

mistake that needs fixing is actually earlier in the program than where Python

*noticed* it was confused. So the line and character that Python indicates in

a syntax error may just be a starting point for your investigation.

**Logic errors** A logic error is when your program has good syntax but there is

a mistake in the order of the statements or perhaps a mistake in how the

statements relate to one another. A good example of a logic error might be,

“take a drink from your water bottle, put it in your backpack, walk to the

library, and then put the top back on the bottle.”

**Semantic errors** A semantic error is when your description of the steps to take

is syntactically perfect and in the right order, but there is simply a mistake

in the program. The program is perfectly correct but it does not do what you

*intended* for it to do. A simple example would be if you were giving a person

directions to a restaurant and said, “. . .when you reach the intersection with

the gas station, turn left and go one mile and the restaurant is a red building

on your left.” Your friend is very late and calls you to tell you that they are

on a farm and walking around behind a barn, with no sign of a restaurant.

Then you say “did you turn left or right at the gas station?” and they say, “I

followed your directions perfectly, I have them written down, it says turn left

and go one mile at the gas station.” Then you say, “I am very sorry, because

while my instructions were syntactically correct, they sadly contained a small

but undetected semantic error.”.

Debugging

**reading** Examine your code, read it back to yourself, and check that it says what

you meant to say.

**running** Experiment by making changes and running different versions. Often

if you display the right thing at the right place in the program, the problem

becomes obvious, but sometimes you have to spend some time to build

scaffolding.

**ruminating** Take some time to think! What kind of error is it: syntax, runtime,

semantic? What information can you get from the error messages, or from

the output of the program? What kind of error could cause the problem

you’re seeing? What did you change last, before the problem appeared?

**retreating** At some point, the best thing to do is back off, undoing recent changes,

until you get back to a program that works and that you understand. Then

you can start rebuilding.

**1.13 Glossary**

**bug** An error in a program.

**central processing unit** The heart of any computer. It is what runs the software

that we write; also called “CPU” or “the processor”.

**compile** To translate a program written in a high-level language into a low-level

language all at once, in preparation for later execution.

**high-level language** A programming language like Python that is designed to

be easy for humans to read and write.

**interactive mode** A way of using the Python interpreter by typing commands

and expressions at the prompt.

**interpret** To execute a program in a high-level language by translating it one line

at a time.

**low-level language** A programming language that is designed to be easy for a

computer to execute; also called “machine code” or “assembly language”.

**machine code** The lowest-level language for software, which is the language that

is directly executed by the central processing unit (CPU).

**main memory** Stores programs and data. Main memory loses its information

when the power is turned off.

**parse** To examine a program and analyze the syntactic structure.

**portability** A property of a program that can run on more than one kind of

computer.

**print function** An instruction that causes the Python interpreter to display a

value on the screen.

**problem solving** The process of formulating a problem, finding a solution, and

expressing the solution.

**program** A set of instructions that specifies a computation.

**prompt** When a program displays a message and pauses for the user to type some

input to the program.

**secondary memory** Stores programs and data and retains its information even

when the power is turned off. Generally slower than main memory. Examples

of secondary memory include disk drives and flash memory in USB sticks.

**semantics** The meaning of a program.

**semantic error** An error in a program that makes it do something other than

what the programmer intended.

**source code** A program in a high-level language.