Helpful Tips for Beginning Computer Science Students

Evan Russenberger-Rosica

Contents

Preface

The view of this text is that the high level goals of an introductory Computer Science course are as follows:

(1) Introducing basic programming concepts
(2) Teaching you how to turn words into code
(3) Learning about the history of computer science
(4) Learning how to Troubleshoot, i.e. helping you to help yourself.

An introductory course is not designed to enable the student to achieve mastery of the subject. The goal is merely to acquaint you with the landscape of the discipline, and to reveal a path forward for the student. The decision whether or not to follow that path, is of course up to the reader.

Style Guide

- Terms in **bold** are new terms being defined.
- The python command prompt is denoted with three rightwards facing arrows as follows

```
>>> python code
```

- In-line Python code is written in the `typewriter` font.

Part 1

Background: Hardware, History, and Math

CHAPTER 1

Hardware

1.1. What is a Computer?

A computer is a device for storing and manipulating information. Information is stored in your computer's **memory** and manipulated by the **processor**. The basic unit of information is known as a **bit**, short for binary digit. The key property of bits is that at any point in time they may represent precisely one of two states. These two states referred to as 0 and 1 or False and True. A group of eight bits is called a **byte**. We will revisit bytes and bits in 3.

A computer **program** is a list of instructions used to control a computer. A **programming language** is set of tools and rules for writing a computer program. A program which follows the semantic and syntactic[1] rules of a programming language is said to be **written in** said language. There are hundreds of programming languages in use today with varying levels of popularity.

Computers only "speak" one language natively; **machine code**. Each machine code instruction is simply a pattern of bits that corresponds to a particular command executed by the processor. Because these commands are executed directly, machine language programs are very efficient and require very little to no overhead. However, the process of writing machine code is error prone and time consuming. Even worse each processor has its own instruction set, meaning code is not portable between different processor models. **Assembly language** improved upon machine code by using a simple mnemonic instead of a number to represent each instruction. A special program known as an **assembler** then translated the assembly to machine code which could then be executed by the processor. Although an improvement, assembly language still possessed many of the same problems as machine language; it was hard to understand, the instructions were so basic that even simple programs could be quite long, and each processor family had its own variant, meaning assembly programs still weren't widely portable.

Fortunately, the incredible growth of computing power has liberated programmers from operating at the level of the machine. In contrast to machine language and assembly, modern programming languages are designed first and foremost so that their programs can be read and understood by humans. This design goal requires that languages hide or suppress some of the underlying complexity of the system in order to simplify it, a process known as **abstraction**.[2] Languages with a high level of abstraction are known as **high-level** languages. Languages with little

[1]Semantic - adj: relating to meaning, Syntactic - adj: relating to form or structure

[2]Philosopher Alfred North Whitehead once said, "Civilization advances by extending the number of important operations which we can perform without thinking about them." Abstraction is the result of applying this philosophy to programming.

to no abstraction, such as machine code and assembly, are known as **low level** languages. We must note that these terms are somewhat relative. Languages such as C, which were once considered high level (relative to assembly), are now considered low level due to the emergence of even higher level languages such as Python.

Regardless of how high level a programming language is, computers still fundamentally operate on bits. This means they require special tools to understand programs written in high level languages. The principal tools are programs known as **compilers** and **interpreters**. A compiler is simply a program for translating programs from one programming language to another; usually from a higher level language to a lower level one. For example, a program written in the the C programming language must first be compiled to assembly, and assembled to machine code before it is executed. In contrast, an interpreter is a program which directly executes programs written in a high level language *without* first translating them into a lower level language. How exactly compilers and interpreters accomplish their tasks is an important but advanced topic in computer science.

1.2. Memory

In this section we will focus on how computers store information. To reiterate:

> The basic unit of computer storage is the bit. A bit can contain one of two values, 0 and 1. All other storage in a computer is based on collections of bits. Given enough bits, it is amazing how many things a computer can represent: numbers, letters, images, movies, sounds, documents, and programs, to name a few. A byte is 8 bits, and on most computers it is the smallest convenient chunk of storage. For example, most computers don't have an instruction to move a bit but do have one to move a byte. A less common term is **word**, which is a given computer architecture's native unit of data. A word is made up of one or more bytes.... A computer executes many operations in its native word size rather than a byte at a time. Computer storage, along with most computer throughput, is generally measured and manipulated in bytes and collections of bytes. A kilobyte, or KB, is 1,024 bytes; a megabyte, or MB, is 1,0242 bytes; a gigabyte, or GB, is 1,0243 bytes; a terabyte, or TB, is 1,0244 bytes; and a petabyte, or PB, is 1,0245 bytes. Computer manufacturers often round off these numbers and say that a megabyte is 1 million bytes and a gigabyte is 1 billion bytes.[**19**, 9]

Computers use several different kinds of memory for different purposes. Each kind of memory is built on an underlying technology, and these vary greatly in performance and price. In general, there is a positive correlation between memory price per bit and performance. This is caused by the simple fact that faster memory is harder to manufacture. A difficult manufacturing process means more units will be defective and cannot be sold. The cost of these defective products is then passed on to the consumer, thereby raising the price of the units that can be sold. Thus an affordable computer can only contain a small amount of the fastest memory. Intelligent use of the limited supply of fast memory means that it is only used to store the most frequently accessed information. Luckily, there are slower, simpler to manufacture memory technologies which enable users to store great quantities of

Name	Technology	Price/GB	Latency
Registers	Custom Circuits	> $1330	< .5ns
Cache	SRAM	$1330	.5 – 2.5ns
Main Memory	DRAM	$9.06	50 – 70ns
Solid-State Memory	NAND Flash Memory	$.25	5,000 – 50,000ns
Hard Drives	Magnetic Disks	$.025	5,000,000–20,000,000ns
Tapes	Magnetic Tapes	$.005	> 20,000,000ns

Table 1. Memory Types, Technologies, and Speed

information inexpensively. Thus memory runs the spectrum from fast, expensive, and small, to slow, inexpensive, and large.The trade-off between memory speed and size means that modern computers rely on several different kinds of memory which we organize into a **memory hierarchy**. [**5**, 375]

From fastest to slowest, the kinds of memory used in modern computers are **SRAM** (static random access memory), **DRAM** (dynamic random access memory), **flash memory**, **magnetic disks**, and **tape**. Memory technologies can be classified into two categories. **Volatile memory**, such as SRAM and DRAM, retains data if and only if it is receiving power. This implies that if volatile memory does not receive power then it does not retain data. In contrast, **nonvolatile memory** retains data even without power. Thus nonvolatile memory, like hard disks, tapes, or even CD/DVDs is used for long term storage.[**5**, 22] One critical memory performance factor is **latency** - the delay between when the processor (the subject of the next sections) requests information from the memory and when it is received.[**19**, 11].

Now that we have discussed the theory behind the memory higherarchy, let's see how/if it works in practice. Using data from Amazon and Digi-Key Electronics we can construct a modern memory hierarchy[3], complete with prices and sizes.[**8, 18, 15, 4, 6**]

We see that for each level of the hierarchy, the price/GB generally increases by at least an order of magnitude (i..e a factor of 10), showing the trade-off between price and speed.

This may seem quite complicated but we can draw an analogy with how our own memory works. The study of how people retain information far predates computer science, and they way computer memory works is partially a reflection of our understanding of human memory. Computers have different kinds of memory, just as humans have short term and long term memory. Psychological studies have shown that the average human can hold about 7 ± 2 objects in short term memory so long as they concentrate on them. This fact is known as Miller's Law, shows humans have a relatively small amount of short term memory, just like a

[3]We have based our our example hierarchy on parts of a size and speed which might be included in a higher end (~$1000-$1500) PC in mid 2018. This sample system uses 2400MHz DDR4 RAM, an Intel 7700k x86-64 processor (based on 12nm lithography) equipped 8MB of L3 SRAM Cache (10 ns latency), a 500GB Samsung V-NAND solid state drive, and an 8TB 7200 RPM SATA3 hard disk drive, as well as a 15GB IBM tape backup drive. Since Intel does not publish exact specifications of the SRAM it uses for its L3 cache, it is difficult to find an accurate price. Therefore, we have presented the per/GB cost of the cheapest SRAM which is the appropriate size (8MB) and latency (10ns). The price of the 7700k's registers would be extremley difficult to discern and is ommitted.

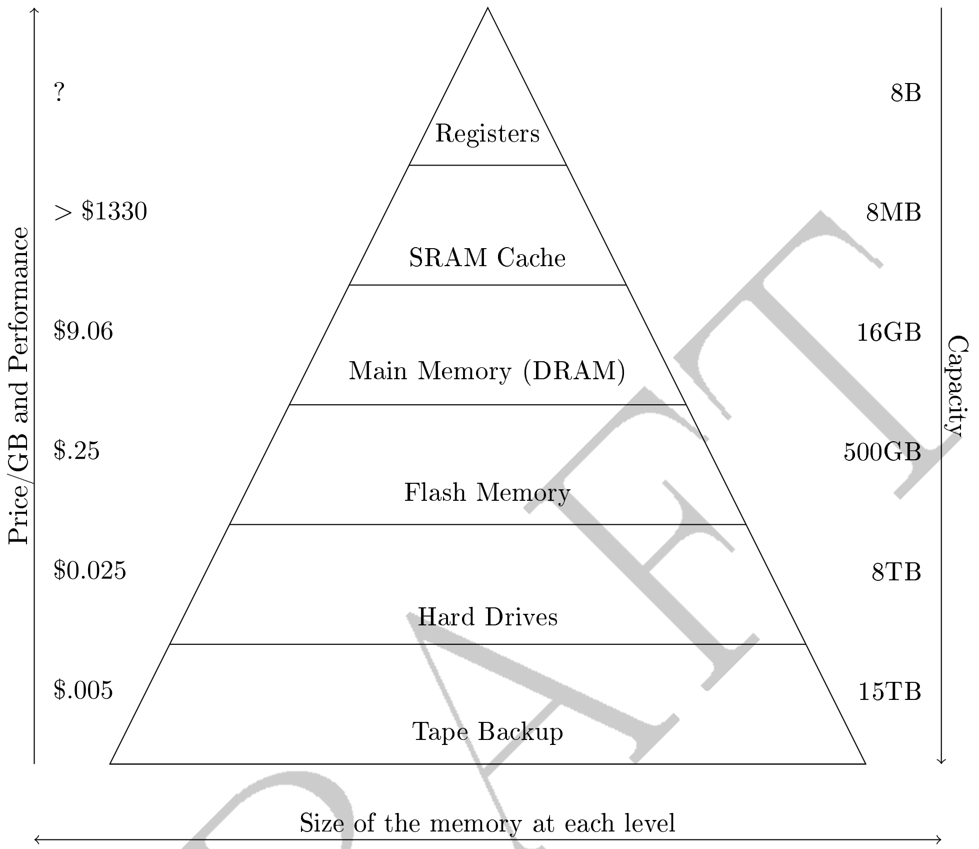

FIGURE 1.2.1. Price/Performance & the Modern Memory Hierarchy (2018)

computer.[**12**] For example, if we concentrate on the numbers 3855202, we can keep them in our short term memory. However, its unlikely we will recall them tomorrow as we have not commuted them to our long term memory. This is similar to how computers can keep information in volatile memory so long as it receives power, but loses the information once power is withdrawn. Furthermore, computers can keep important, frequently accessed memory in cache, much like a person might write on a notecard.

In summary, the essence of the memory hierarchy is that computers have a little bit of very fast, very expensive memory, and a lot of slower, less expensive memory. Important, frequently accessed files are kept in the fast memory in a process known as caching. Since the fastest memory is volatile, information must be saved in slower nonvolatile memory to be preserved. For more detailed information on computer memory, see [**5**, Chapter 5].

1.3. Processor

Earlier, we said that a computer is a device for storing and manipulating information. In this section, we focus on the device which manipulates information, the **processor**. TODO

1.4. Motherboards

How all these pieces connect and talk to each other

1.5. Chapter Summary

Terms

- Computer: “a device for storing and manipulating information”
- Memory: “Where information in a computer is stored”
- Processor: “Manipulates information in a computer”
- Bit: Basic unit of information. Has two states: 1 and 0.
- Byte: 8 bits
- Program: A list of instructions for the computer
- Programming Language: a set of tools and rules for writing a computer program
- Machine code: the only native language of computers
- Assembly language: improved on machine code by using a mnemonic instead of a number to represent instructions
- Assembler: A program which translates assembly to machine code
- Abstraction: the process of hiding complex details of a system in order to simplify it
- Compiler: a program for translating programs from one programming language to another; usually from a higher level language to a lower level one.
- Interpreter: a program which directly executes programs written in a high level language

CHAPTER 2

Basic CS History

2.1. Babbage Difference Engine

2.2. Computers, Cryptography, WWII

2.2.1. Alan Turing.

2.2.2. Turing Tests.

2.2.3. Turing Machine.

2.2.4. Bletchley Park, Bombe, Colossus.

2.3. Von Neumann

2.4. ARPANET → W.W.W.

2.5. Unix Development → C → Python

CHAPTER 3

Some Math

Computers appreciate precision; they are not at all good at improvising. Since mathematics is the language of precision, we will begin with a quick review of some key concepts.

3.1. Some Logic

Logic is a metalanguage for math. Before we dive into math, we need a language to discuss it; logic fills this role.

Definition 1. In mathematics, a **proposition** or **statement** is a "declarative sentence that is either true or false but not both. The key is that there must be no ambiguity. To be a statement, a sentence must be true or false, and it cannot be both". [**21**, 1]

Example 2. A sentence such as "The sky is beautiful" is not a statement since whether the sentence is true or not is a matter of opinion. A question such as "Is it raining?" is not a statement because it is a question and is not declaring or asserting that something is true." [**21**, 1]. The sentences $1 + 1 = 2$ and $1 + 1 = 3$ are examples of true and false statements respectivley.

Definition 3. A **variable** is a symbol representing an unspecified object that can be chosen from a given set U. The set U is called the **universal set** for the variable. It is the set of specified objects from which objects may be chosen to substitute for the variable. A **constant** is a specific member of the universal set.[**21**, 54]

Variables create an interesting dilemma. What are we to make of sentences like $1 + x = 2$? Is this a statement? Since the truth of the sentence depends on the value of x; $1 + x = 2$ is true for some values of x and not for others. Without knowing x, $1 + x = 2$ is neither true nor false, and thus it is not a statement. To create a statement from such an expression, we need another tool; the predicate.

Definition 4. A **predicate** or a **propositional function** is a sentence $P(x_1, x_2, \dots, x_n)$ involving variables $x_1, x_2, \dots, x_n$ with the property that when specific values from the universal set are assigned to $x_1, x_2, \dots, x_n$, then the resulting sentence is either true or false. That is, the resulting sentence is a statement.[**21**, 56]

Example 5. Let the universal set $U = \mathbb{R}$, and let $P(x)$ be the predicate $x^2 = 4$. Then:

(1) Then substituting $x = 2$ into the predicate gives the true statement $2^2 = 4$
(2) Substituting $x = \sqrt{2}$ into the predicate gives the false statement $2 = 4$

EXAMPLE 6. Let the universal set $U = \mathbb{Z}$, and let $P(x_1, x_2, x_3)$ be the predicate $x_1^2 + x_2^2 = x_3^2$. Then:

(1) Substituting $x_1 = 3, x_2 = 4, x_3 = 5$, gives the true statement $3^2 + 4^2 = 5^2$. In geometry, any three integers which together satisfy this equation are known a Pythagorean Triple and constitute valid side lengths of a right triangle.
(2) Substituting $x_1 = 4, x_2 = 5, x_3 = 6$, gives the false statement $4^2 + 5^2 = 6^2$. Thus $(4, 5, 6)$ are not valid side lengths of a right triangle.

DEFINITION 7. The **truth set** of a predicate is the collection of objects in the universal set U that can be substituted for the variable to make the predicate a true statement.

EXAMPLE 8. Let $U = \mathbb{N}$ and $P(x)$ be the predicate $x^2 = 4$. Then the truth set of $P(x)$ is $\{2\}$. If we let $U = \mathbb{Z}$, then the truth set of $P(x)$ is $\{-2, 2\}$.

3.2. Introduction to Sets

A **set** is a collection of objects, which we call its **members**, **elements**, or **points**. These objects could be almost anything, numbers, people, planets, etc. We denote that a is a member of set A by $a \in A$, and likewise that a is not a member by $a \notin A$.

There are two common set notations. Firstly, we have the **roster method** which specifies a set as a list of comma separated elements enclosed in curly brackets. For example, the set containing the numbers $1, 2, 3$ is written $\{1, 2, 3\}$. The roster method can also be used to denote large (or even infinite) sets with a clear pattern. In this case, we list several elements (hopefully enough to illustrate the pattern), then use the 3 dots symbol $(\ldots)$ to indicate that the pattern continues. For example, we could write the powers of 2 as $\{1, 2, 4, 8, 16, \ldots\} = A$, and the reader could deduce that each $x \in A$ equals 2^n for some integer n. However, having to guess the pattern every time we deal with a set is tedious, error prone, and not always possible. This begs the question, if a set is defined by a rule or property, why not simply make that property explicit? This is the motivation behind **set-builder notation**. [1] in which each set A is explicitly defined in terms of a rule, or more precisely a predicate $P(a)$, that all elements a of A must satisfy. In set builder notation, the predicate acts as an entrance test; $a \in A$ if and only if $P(a)$ is true. The set-builder notation for the set of all objects x in universal set U which satisfy the predicate $P(x)$ (i.e. the truth set of $P(x)$ in U) is

$$\{x \in U : P(x)\}$$

If the universal set is clear from context, we will omit it and write $\{x : P(x)\}$. This is usually read as "the set of all x such that $P(x)$" with the vertical bar standing for "such that". Some writers use a colon (:) instead of a vertical bar.

EXAMPLE 9. Set Builder Notation

[1] To be precise, we could define **set comprehension** as the process or idea of defining a set by a property and set-builder notation as the syntax for describing a set so defined, however most authors seem to consider set comprehension and set-builder notation to be synonymous and so will we.

(1) Let U be the set of all presidents of the United States. If $P(x)$ is the property that x had a child who became president then $\{x \in U : P(x)\}$ is the set of all presidents who had a child who also became president.

$$\{x \in U : P(x)\} = \{\text{John Adams, George H.W. Bush}\}$$

(2) Let U be the set of all baseball players. If $P(x)$ is the property that x hit more than 700 career home runs, then $\{x \in U : P(x)\}$ is the set of all baseball players who it more than 700 career home runs and:

$$\{x \in U : P(x)\} = \{\text{Babe Ruth, Hank Aaron, Barry Bonds}\}$$

Some sets are so common that they have special names and symbols which are mostly standardized:[2]

Name	Symbol	Contents	Heuristic
Natural Numbers	$\mathbb{N}$	$\{0, 1, 2, 3, 4, 5, \dots\}$	Non-Negative Integers
Positive Integers	$\mathbb{N}^\star$	$\{1, 2, 3, 4, 5, \dots\}$	Positive Integers
Integers	$\mathbb{Z}$	$\{\dots, -2, -1, 0, 1, 2, \dots\}$	Whole Numbers
Rational Numbers	$\mathbb{Q}$	$\{a/b : a, b \in \mathbb{Z}^\star\}$	Fractions ($\mathbb{Z}^\star = \mathbb{Z}\backslash\{0\}$)
Irrational Numbers	$\mathbb{R} - \mathbb{Q}$	$\{x \in \mathbb{R} : x \notin \mathbb{Q}\}$	Can't be written as a fraction
Real Numbers	$\mathbb{R}$		All non complex numbers

TABLE 1. Common Sets

We can visualize the relationships between these important sets using an Euler diagram below.

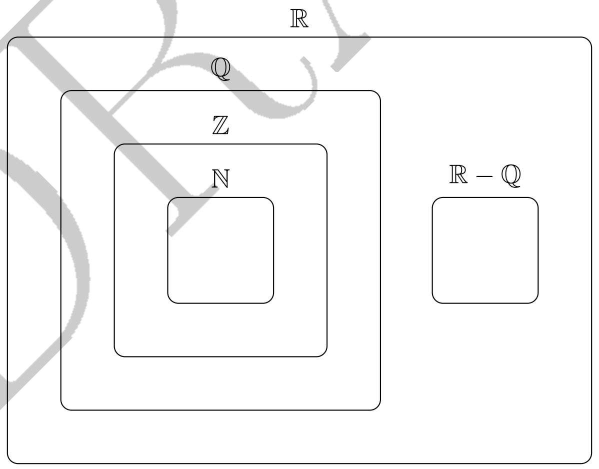

FIGURE 3.2.1. Subsets of the Real Numbers

Let's look at some examples of sets written in set builder notation:

[2]There has been much historical debate over the inclusion of 0 in the natural numbers. Our notation follows the convention of ISO 80000-2 by including 0 in $\mathbb{N}$, and using the notation $\mathbb{N}^\star$ to refer to the positive integers [**9**, p. 6]. Since this convention is relativley recent, the reader is likely to encounter differing notation elsewhere.

Example 10. The set of all even numbers can be written in set builder notation as $\{x : x \text{ is even}\}$. By definition, a number x is even if there is some integer k such that $x = 2k$. Thus we have

$$\{x : x \text{ is even}\} = \{x : x = 2k \text{ for some } k \in \mathbb{Z}\}$$

Thus $P(x)$ is the statement that "$x = 2k$ for some $k \in \mathbb{Z}$" and $P(x)$ is true for $x = 0, 2, -2, 4, -, 4, \cdots$. Thus the truth set of $P(x)$ is $\{x : x = 2k \text{ for some } k \in \mathbb{Z}\} = \{\ldots, -4, 2, 0, 2, 4, \ldots\}$.

Definition 11. Subset

Suppose A and B are sets. If every object that belongs to A also belongs to B then we say "A is a **subset** of B" or "A is contained in B" written $A \subseteq B$. Symbolically, $A \subseteq B \iff \forall a(a \in A \to a \in B)$. Likewise we write $A \not\subseteq B$ to mean $\neg(A \subseteq B)$, i.e. A is not a subset of B.

Definition 12. Collection

It is very common to have a set whose elements are themselves sets. For example $\{\{0,1\},\{1,2\}\}$ is a set with two elements $\{0,1\},\{1,2\}$ both of which are sets. We call such sets **collections** and denote them as bolded capital letters, $\boldsymbol{A, B, C, \ldots, X, Y, Z}$. Observe that all collections are sets.

Definition 13. Set Equality

Two sets X, Y are equal iff they have the same elements. This means for any sets X, Y, $X = Y$ iff $X \subseteq Y$ and $Y \subseteq X$.

From this definition we can deduce four important properties of sets:

(1) Sets are unordered: $\{1,2,3\} = \{3,1,2\}$ since they have the same elements.
(2) Repeat elements are ignored. Thus for instance $\{1,2,3\} = \{1,1,3,2\}$ because all of these sets have the same elements.
(3) Sets with different predicates may be equal: $\{x : 3x + 6 = 0\} = \{x : x + 2 = 0\} = \{-2\}$ since all have the same element: -2.
(4) Sets with same predicate may not be equal if their universal sets are different: $\{x \in \mathbb{N} : x^2 = 4\} = \{2\}$, but $\{x \in \mathbb{Z} : x^2 = 4\} = \{-2, 2\}$.

Example 14. $\{x : x^2 + 2x = 0\}$ and

Definition 15. The set with no elements is called the **empty set**, and is denoted $\emptyset$, or $\{\}$.

Example 16. Prove that the Empty Set $\emptyset$ is a subset of every set Y.

Proof. By the definition of 11, to show that a given set X is a subset of a set Y, we must show that every element of X is also an element of Y. Since the empty set has no elements, it is vacuously true that every element of $\emptyset$ is contained in Y. □

3.3. Set Operations

In arithmetic, we manipulate numbers via the operations of addition, subtraction, multiplication, and division. In this section we will introduce similar operations for manipulating sets.

DEFINITION 17. The **union** of A and B written $A \cup B$ is the set of all objects which are members of A or B. The **intersection** of A and B written $A \cap B$ is the set of all things that are members of both A and B. Sets A and B are called disjoint iff their intersection is empty, i.e. they have no members in common. A collection of sets is **pairwise disjoint** iff any any two elements of the collection are disjoint. Formally, we can define the union and intersection of arbitrary amounts of sets as follows; Let $\boldsymbol{A}$ be a collection. Then the union of $\boldsymbol{A}$, $\bigcup \boldsymbol{A}$ is the set consisting of every x such that x is belongs to some member of $\boldsymbol{A}$. Similarly, $\bigcap \boldsymbol{A}$ is the set consisting of every x such that x belongs to all members of A. Formally, we have:

$$\bigcup \boldsymbol{A} = \{x : x \in A \text{ for some } A \in \boldsymbol{A}\}$$
$$\bigcap \boldsymbol{A} = \{x : x \in A \text{ for every } A \in \boldsymbol{A}\}$$

EXAMPLE 18. If $\boldsymbol{A} = \{\{0,1\},\{1,2\}\}$, $\bigcup A = \{0,1,2\}$, $\bigcap A = \{1\}$. If the collection $\boldsymbol{A}$ only contains a few elements, say A, B, C, we can use the infix notation to denote the union and intersection of these elements as $A \cup B \cup C$ and $A \cap B \cap C$ respectively.

EXAMPLE 19. Let $\boldsymbol{A} = \{\{0,1\},\{1,2\}\{3\}\}$. Then $\bigcup \boldsymbol{A} = \{0,1,2,3\}$.

If two sets intersect (i.e. have common elements), we may wish to discuss the elements which are contained in one set and not in the other. For example, the odd numbers are the elements of $\mathbb{Z}$, which are not even, i.e: $\{x : x \in \mathbb{Z} \wedge x \text{ is not even}\} = \{x : x \in \mathbb{Z} \wedge \neg(\exists n \in \mathbb{Z}(x = 2n))\}$. To express this notion more succinctly, we have the following

DEFINITION 20. Set Difference: $B \backslash A$
The **set-theoretic difference** of B and A (also known as the **relative complement of** A in B) is the set of elements in B but not in A, written

$$B \backslash A = \{x : x \in B \wedge x \notin A\}$$

Although this notation is standardized in [**9**, 5], some authors write $B - A$.

3.4. Relations

DEFINITION 21. We saw earlier that sets are unordered, meaning for example $\{1,2,3\} = \{3,1,2\}$. However, this is not always desirable. Thus we introduce the idea of an **ordered pair** of objects defined so that

$$(x, y) = (u, v) \iff x = u \wedge y = v$$

One way to do this, known as the Kuratowski Formalization, is to define ordered pairs in terms of sets as follows:

$$(x, y) = \{\{x\}, \{x, y\}\}$$

More generally, ordered n-tuples are defined recursively for $n > 1$ by:

$$(x_1, \ldots, x_{n+1}) = ((x_1, \ldots, x_n), x_{n+1})$$

DEFINITION 22. Given two sets A, B, their **Cartesian Product** $A \times B$ is given by $\{(a, b) : a \in A \wedge b \in B\}$. Observe that the notation (a, b) denotes an *ordered* pair, and is thus distinct from the set $\{a, b\}$ which is unordered. For example,

$\{a,b\} = \{b,a\}$, but $(a,b) \neq (b,a)$. The Cartesian product of a set A with itself is written A^2.

stuff

Definition 23. If A and B are sets, a subset R of the Cartesian product $A \times B$ is called a **relation between** A and B. The statement $(x,y) \in R$ is read "x is R-related to y, and is denoted xRy, $R(x,y)$, or $(x,y) \in R$. A relation $R \subseteq A \times A = A^2$ is called a **relation on** A. In general A^n is the set of all n-tuples of members of A. It is important to note that order matters, if $a \neq b$, then aRb does not imply bRa.

Example 24. Let

$$A = \{\text{Chevrolet, Honda, Toyota, Ford}\}$$
$$B = \{\text{Taurus, Corolla, Civic, Corvette}\}$$

Define $R = \{(a,b) \in A \times B : a \text{ manufactures } b\}$. Then $R \subseteq A \times B$ is a relation and

$$R = \{(\text{Ford, Taurus}), (\text{Toyota, Corolla}), (\text{Honda, Civic}), (\text{Chevrolet, Corvette})\}$$

Definition 25. The **domain** of a relation R, written dom$R$ is the set of all objects x such that $(x,y) \in R$ for some y. The **image** (or **range**) of R, written im$R$ (or ran$R$) is the set of all objects y such that $(x,y) \in R$ for some x. Symbolically:

$$\text{dom}R = \{x : (x,y) \in R\}$$
$$\text{im}R = \{y : (x,y) \in R\}$$

3.5. Functions

In elementary school, we are told to think of a function f from a set A into a set B as a rule which transforms a given input x in A into a unique output $f(x)$ in B. The notation $f(x)$ (read f of x) is intended to show that the element $f(x)$ is the result or image of function f acting on element x. The key property of functions is that they are **single-valued**; each input x has a unique output $f(x)$. For instance, in the following figure, we see a relation which is not a function because the input x_2 has two outputs,

However, as we progress to more abstract mathematics, we will need the more precise definition of a function given below.

Definition 26. Let A, B be sets. A relation $F \subseteq A \times B$ is a **function from** A **into** B if: for all $a \in A$ and all $b_1, b_2 \in B$, it is always true that aFb_1 and aFb_2 implies $b_1 = b_2$. We then say that F **maps** A **into** B and write $F : A \to B$. For functions, we use the notation $F(a) = b$ instead of aFb or $(a,b) \in F$. In this notation, we can restate the definition of a function to say that F is a function iff for all $a \in A$ it is the case that $F(a) = b_1$ and $F(a) = b_2$ implies $b_1 = b_2$.

In words, this means no one input may have two distinct outputs.

Some functions have important properties which we will discuss below. If for every $b \in B$ there exists and $a \in A$ such that $b = f(a)$, then f is said to be a **surjection** or **onto**. If for every $a_1, a_2 \in A$ it is the case that $f(a_1) = f(a_2) \to a_1 = a_2$ then f is said to be an **injection** or **one-to-one**. A function which is both onto and one-to-one is said to be **bijective**, or a **bijection**.

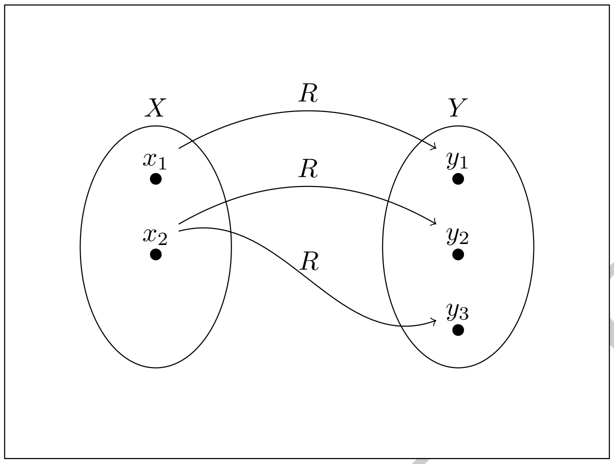

FIGURE 3.5.1. The relation R is not a function

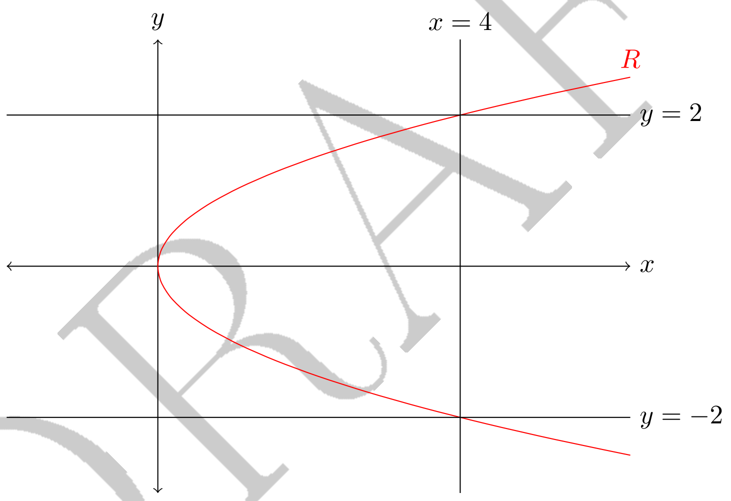

FIGURE 3.5.2. Relation R fails the Vertical Line Test

For instance, in 3.5.2, we see a relation R which is not a function of x because the input $x = 4$ has two outputs $y = 2$ and $y = -2$

Geometrically, this means functions must pass the *vertical line test.*

This is not a function of x because

DEFINITION 27. The **modulo** operation $x \mod y$ returns the remainder of the euclidean division of x by y.

EXAMPLE 28. $5 \mod 3 = 2$ since $5 = 3 \cdot 2 + \underbrace{1}_{\text{remainder}}$. $-2 \mod 3 = 1$ since $-2 = 3 \cdot (-1) + 1$.

DEFINITION 29. In combinatorics, a bijection from a set to itself is sometimes called a **permutation**. Informally, a permutation of a set of objects is just a rearrangement, for instance, $0, 1, 2, 3$ is just a rearrangement of $1, 2, 3, 0$. We see

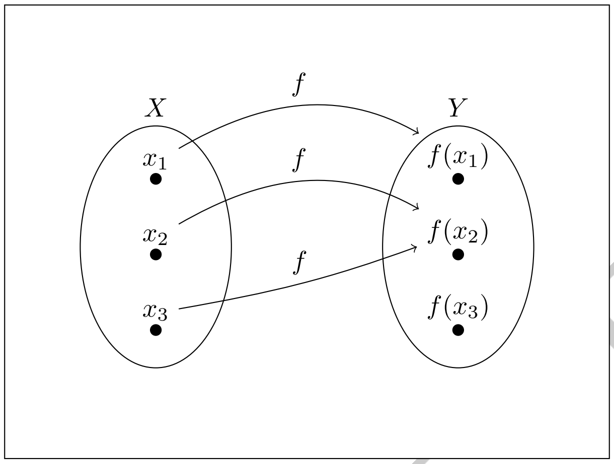

FIGURE 3.5.3. A non-surjective and non-injective function f from X into Y

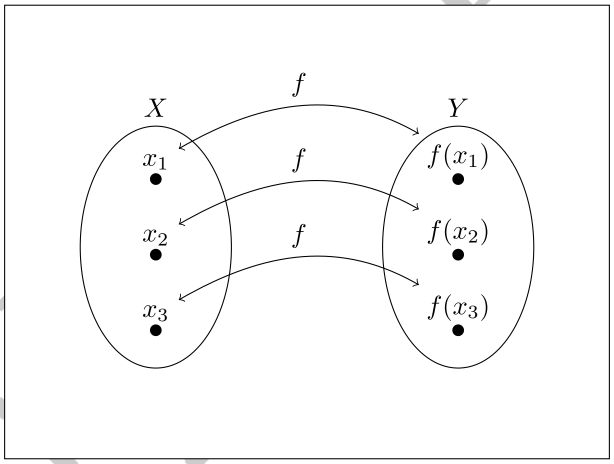

FIGURE 3.5.4. A bijection f from X onto Y

how this is accomplished. Let $X = \{1, 2, 3, 4\}$, and let $f : X \to X$, $x \mapsto x + 1 \text{mod} 4$. Note: $(3 + 1) \mod 4 = 0$.

$$f : \{0, 1, 2, 3\} \to \{0, 1, 2, 3\}$$

3.6. Sequences

DEFINITION 30. A function $a : N \to X$ from a subset of the integers N into a set X is called a **sequence in** X . Although sequences are simply a type of function, there is some special terminology and notation which is unique to sequences. The domain of a sequence is known as its **index set**, and an element of the index set is known as an **index**. The notation a_n is used instead of $a(n)$ to denote the image or value of n under a. We then call a_n a **term** or **element** of the sequence. We use the notation $(a_n)_{n\in N}$ to refer to the sequence itself. If the domain is clear, we may simplify and write (a_n). In summary, if $a : N \to X$ is a sequence, then for each $n \in N$, $a(n)$ is denoted a_n and a itself is denoted $(a_n)_{n\in N}$ or (a_n) for short. [**14**, 156]

The most common domains for finite sequences are sets of the form $\mathbb{N}_N = \{n \in \mathbb{N} : n < N\} = \{0, 1, 2, 3, \ldots, N-1\}$, sometimes called **initial segments** or **sets of strictly preceding elements** of the natural numbers.[**3**, 16] When the domain of a sequence is an initial segment, we can refer to a_n as the nth term of the sequence

EXAMPLE 31. We can define a particular sequence by giving a formula for the nth term a_n. Let (a_n) be the sequence with domain $N = \{1, 2, 3, 4, 5\}$ and whose nth term is $a_n := \frac{1}{n^2}$. Then $(a_n)_{n\in N} = (\frac{1}{1^2}, \frac{1}{2^2}, \frac{1}{3^2}, \frac{1}{4^2}, \frac{1}{5^2}) = (1, \frac{1}{4}, \frac{1}{9}, \frac{1}{16}, \frac{1}{25})$, and we would say 1 is the 1st term or element of the sequence, $\frac{1}{4}$ is the 2nd term, $\frac{1}{9}$ is the third and so on. [**1**, 55]

3.6.1 shows how a sequence (a_n) maps element n of the index set $\{1, 2, \ldots\}$ to the element a_n, which we call the nth term of the sequence.

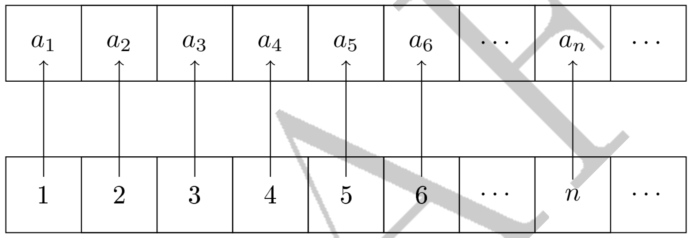

FIGURE 3.6.1. The terms a_n of the sequence (a_n)

3.7. Binary Math

We started 1 by introducing the bit as the basic unit of information is known as a **bit**. We now explore bits and binary arithmetic in more detail. To reiterate, a bit may represent precisely one of two values at any given time. Physically, these two values may be represented by two levels of electric charge stored in a capacitor. two values of a bit may be represented by two levels of electric charge stored in a capacitor.. As we add more bits, we can represent more states. Three bits for example may represent 8 distinct states as shown below.

$$\begin{array}{cccc} 000, & 001, & 010, & 100 \\ 011, & 110, & 101, & 111 \end{array}$$

In general, if we have n objects, each of which can take on m values, there are $n \cdot m$ possible outcomes. This is a fundamental result in combinatorics known as the **rule of product** or **multiplication principle**. Thus a string of n bits which can each represent 2 states creates $2n$ unique bit strings. Each of these strings may be used to represent a different value. A bijective mapping from a set of bit strings to a set of values is known as a **character encoding**. Examples include ASCII and Unicode. (See appendix A). Programmers are able to use these encodings to do some neat tricks. For example, we may convert a lowercase character to its uppercase equivalent as follows:

```
>>> def LowertoUpper(char):
        return chr(ord(char)-32))

>>> LowertoUpper("a")
        'A'
```

Lets examine why this works. The ASCII encodes $01000001_2 = 65_{10}$ to a `A` and $01100001_2 = 97_{10}$ to `a`. This lets C programmers

Traditionally, we denote that a number x is binary by prepending the prefix $0b$. Thus the binary number 101 (read one zero one) is distinguished from the decimal 101 (one hundred and one) by writing 0b101. Appending the base of the number as a subscript is also common: e.g: $101_2 = 5_{10}$.

To understand binary, we must first understand how numbers work, specifically the **positional** or **place-value notation** we use to represent numbers. This notation writes numbers as a sum of increasing powers of 10.

$$\begin{aligned} 27182 &= 2 \cdot 10^4 + 7 \cdot 10^3 + 1 \cdot 10^2 + 8 \cdot 10^1 + 2 \cdot 10^0 \\ &= 2 \cdot 1000 + 7 \cdot 100 + 1 \cdot 100 + 8 \cdot 10 + 2 \cdot 1 \\ &= 2000 + 700 + 100 + 80 + 2 \end{aligned}$$

Generally given a **radix** or **base** b and a sequence of n numbers $(a_n) = a_{n-1}, \ldots, a_1, a_0$, where $0 \leq a_i < b$, their decimal representation is d is:

$$d = \sum_{i=1}^{n} a_{n-1} b^{n-1}$$

3.7.1. Floating Point Arithmetic. Because computers have finite amounts of memory, it is impossible for them to represent numbers which do not have finite base two expansions.

EXAMPLE 32. The number $\frac{1}{3}$ cannot be represented as a finite sequence of digits in base 10. Thus we must express $\frac{1}{3} = .33333\overline{3}$ with the understanding that the 3s go on indefinitely. However, in

3.8. Algebra

Part 2

Programming Concepts

CHAPTER 4

Variables

Python is a (very) high level language. This means there is a great deal of abstraction between what the machine does, and what the programmer actually sees. While this has numerous benefits, including ease of programming, it means that there are key concepts which are difficult to examine in Python because they are taken care of automatically by the interpreter. In other words, in order to be intuitive to the user, Python does a lot in the background. To learn more, we must examine a language which is less intuitive to humans, but simpler from the perspective of the computer - the C programming language. For a brief introduction, we turn to the canonical C reference [**2**, 8];

> C is a general-purpose programming language with features economy of expression, modern flow control and data structures, and a rich set of operators. C is not a "very high level" language, nor a "big" one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages. C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie. The operating system, the C compiler, and essentially all UNIX applications programs (including all of the software used to prepare this book) are written in C. ... C is not tied to any particular hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C.

4.1. In the Beginning - C Style Variables

Conceptually, we can think of a **variable** as a box or container for information. Your computer stores this box in its short term memory. Where exactly in memory a variable is stored is refereed to as a variable's **address**. Every byte of memory has its own unique address, usually represented as a hexadecimal number like `0x7fff1be48dd4`. The amount of memory space a variable occupies is its **size**. When a variable `x` is declared, the compiler sets aside n bytes of contiguous space in memory to hold `x`. That is to say, if the size of `x` is 10 bytes, and `y` is a memory address, the compiler sets aside addresses `y`,`y+1`,`y+2`$\cdots$`y+9` to hold `x`. The kind of information contained in a variable is known as its **type**. Examples of types include the character type, used to store a single character, such as `a`, from among the 256 ASCII characters, and the integer type which can store any integer between -2,147,483,648 and 2,147,483,647. Some variables take up more room and thus require a larger box in memory to contain them. For instance `a` takes up less room than `2,147,483,647`. For this reason, the amount of memory the C compiler sets aside for each variable we declare is directly determined by its type.

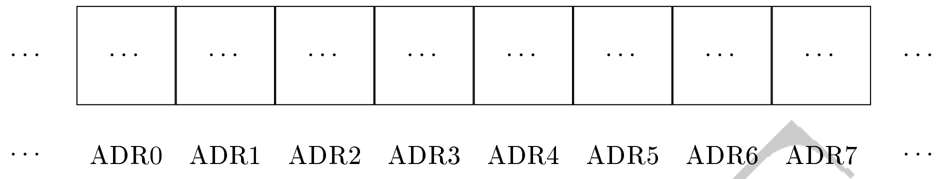

FIGURE 4.1.1. Memory Array

4.1.1. Assignment and Declaration. In C, all variables must be declared before they are used, usually at the beginning of the function before any executable statements. A **declaration** announces the properties of variables; it consists of a type and a list of variables, such as

```
int fahr, celsius;
float step;
```

The type `int` means that the variables listed are integers; `float` means that the variables are floating point numbers, i.e., numbers that may have a fractional part. A language like *C* in which the programmer must explicitly identify the type of a variable at declaration is said to be **manifestly** or **explicitly typed**.

Once we declare our variables, the compiler allocates memory for them. The amount of memory a variable is allocated depends on its type. If we suppose floats occupy three bytes in memory, ints occupy two, and three dots $\cdots$ represent empty space, then after declaring the variables `fahr`, `celsius`, and `step` the memory of the computer looks like figure 4.1.2. Note how the integer variable `fahr` only takes up two physical bytes on memory while `step` takes up three physical bytes because it's of type `float`. Also note that while each variable is made up of a contiguous block of bytes, the variables need not be contiguous (note the unused space at `ADR2` between `fahr` and `step`).

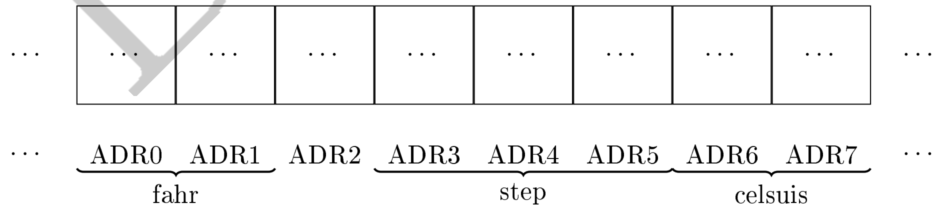

FIGURE 4.1.2. Physical Memory Array after Declaration

At this stage, we have declared our variables and set aside memory to hold them, however, we have not yet assigned them any value. In order to assign value to our variables, we must use an **assignment statement**. In C, assignment statements look as follows:

```
fahr = 2,147,483,647;
celsius = 314159265358.979323;
step = −2,147,483,647;
```

Note that once a variable is declared, we do not need to state its type again during assignment. E.g, if we have previously declared `float lower;`, we simply write `lower = 20;` to assign a value to `lower`. The first time a variable is assigned a value is called **initialization**, and a variable which has undergone initialization is said to be **initialized**.[1][**16**, 233] C allows us to declare and initialize a variable all in one line, with the syntax `int x = 1;`.Following these assignment statements, our memory might look something like this:

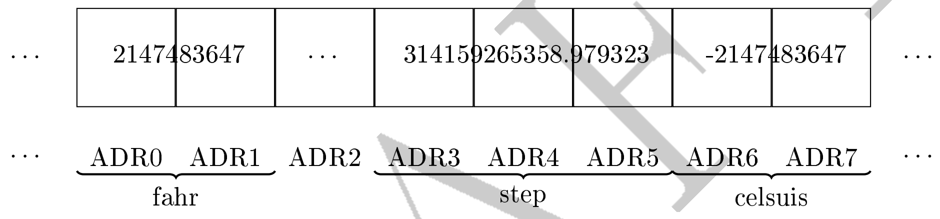

FIGURE 4.1.3. Physical Memory Array after Initialization

4.1.2. Pointers. A computer has a set of memory addresses which label the bytes of its physical memory. Computer programs store their data in variables, which are an abstraction of this physical memory. In certain situations, it is advantageous to take go behind the scenes and directly manipulate the relationship between variables and their address. This is the function of pointers.

A **pointer** `p` is a variable that contains the address of a variable `c`. The unary **referencing operator** `&` is used to access the address of an object `c`. Thus the statement `p=&c;` assigns the address of `c` to the variable `p`. If `p` is a pointer to `c`, we say `p` "points to" or is a "reference to" `c`. We may also refer to `c` as the "pointee" of `p`. The unary operator `*` is the indirection or **dereferencing operator**; when applied to a pointer, it accesses the object the pointer points to.

We can declare pointers similarly to how we declare variables. The code `int = *ip;` declares that `*ip` is an integer, and therefore that `ip` is a pointer to an integer. This notation is indented to suggest that `ip` points to an integer `x`, then `*ip` can occur anywhere `x` could. Pointers are constrained to point to a particular kind of object: every pointer points to a specific data type.[**2**, 78].

EXAMPLE 33. From [**2**, 78]. Consider the following code:

[1]Surprisingly, C allows us to reference variables which have been declared but not initialized. E.g., `int x; x+1;` is valid C code which declares a variable `x` and simply adds 1 to *whatever value* was contained at the address of `x`. The value of an uninitialized variable is said to be **undefined** or **garbage**.[**2**, 72] Unintentional use of undeclared variables is a frequent cause of errors.

```
int x=1, y, *ip;     /* declare ip is a pointer to an int */
ip = &x;            /* assign ip the address of x */
y = *ip;            /* y is now 1 */
```

Lets break this code down line by line:

(1) We declare two integers `x`,`y`, initialize `x` to 1, and a declare `ip` is a pointer to an integer (`ip` short for **i**nteger **p**ointer).
(2) `ip = &x;` assigns `ip` the address of `x`, so that `ip` points to or references `x`.
(3) The assignment statement `y = *ip;` sets `y` equal to the value stored at the address contained in `ip`. Since line 3 told `ip` to store the address of `x`, we have that `y` is equal to the value stored at the address of `x`, which is `1`.

We depict the state of the program after the second line in figure 4.1.4. To simplify, suppose each variable is only one byte, so our program only involves 3 bytes and 3 addresses. `ADR1` is the address of the pointer `ip`, and `ADR2` and `ADR3` are the addresses of `x`,`y` respectively. At this stage of the program, our variables and pointer have been declared, and `ip` has been initialized to point to `x`, which we symbolize with an arrow form the address of the pointer `ip` to the address of `x`. That is, the address of `x` has been stored at `ADR1` as the value of `ip`. To evaluate `y = *ip;`, we follow the arrow from `ADR1` to `ADR2` and save the value stored at `ADR2` to `y` in `ADR3`. That is, to evaluate `y = *ip;`, we go to the address (`ADR2`) stored in `ip` , and save the value stored at that address to `y` at `ADR3`. I.e., in `y = *ip;`, we deference `ip`, and save the result to `y`.

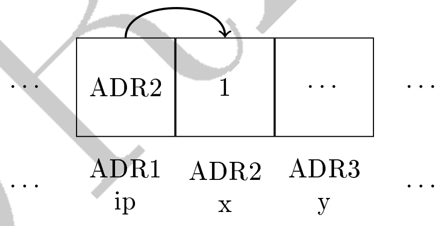

FIGURE 4.1.4. `ip=&x`

Pointer Assignment & Shared References. Pointer assignment between two pointers makes them point to the same pointee. So the assignment `y = x`; makes `y` point to the same pointee as `x`. Pointer assignment does not touch the pointees. It just changes one pointer to have the same reference as another pointer. After pointer assignment, the two pointers are said to be "sharing" the pointee, or that they have a **shared refrence**. In the following code, we declare two pointers `p1`, `p2`. We then set `p1` equal to the address of `x` via the referencing operator `&`. Finally, we use pointer assignment so that `p1=p2=&x`. We dereference `p1 and p2` and print the results.

In memory, a shared refrence might look something like this:

Algorithm 1 Creating a Shared Refrence

```
int x=1, y=2 *p1, *p2;
p1=&x;
p2=p1;     //pointer assignment creates a shared refrence
printf("%p\t%p\n",*p1,*p2);

>>> 1   1
```

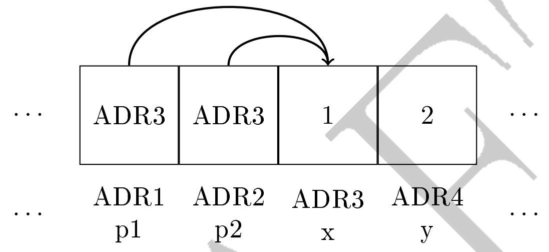

FIGURE 4.1.5. Shared Refrence

Shared refrences are a powerful tool. If `p1`, `p2` are pointers to `x`, then changing the value of `x` changes the value of `*p1`, `*p2`. (Why does changing the value of `x` affect `*p1`, `*p2`; but not the value of `p1` and `p2`?)

EXERCISE 34. Suppose that after the line `p2=p1;`, we assign `p1=&y`. What is the output of 1? Draw a diagram representing the situation

The statement `p1=&y` simply tells saves the address of `y` in `p1`, or more precisley in the memory address of `p1`. Nothing else changes. Thus `p2` still points to `x`, and thus `*p2` equals `1`, while `p1` points to `y`, and thus `*p1` equals `2`. We can represent the situation pictorially below. Note that figure 4.1.5 and figure 4.1.6 represent the same code before and after the assignment `p1=&y`.

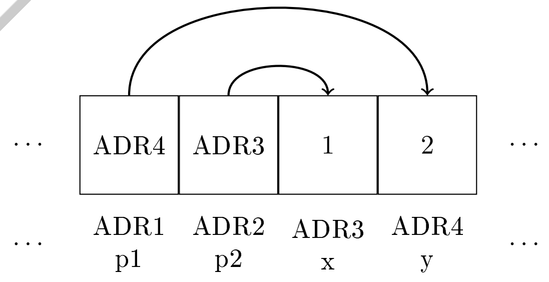

FIGURE 4.1.6. Reassigning a Shared Reference

It is important to note that the pointer assignment p2=p1 is not the same as stating that p2 is a pointer to p1. In the example above we showed that if p1,p2 have a shared reference to x, then reassigning p1 did not effect p2. The key realization is that the statements

```
p2=p1
p1=&y
```

are assignments not equalities. We must fight our mathematical impulse to apply the transitive property and say that $p2 = p1$ and $p1 = \&y$ implies $p2 = \&y$. It only matters what the p1 pointed to when we assigned p2=p1, as this is the address that is stored in p2. Our subsequent reassignment of p1 has no bearing on p2. Put succinctly, assignment does not create a chain of pointers. Do not read the above code as saying p2 points to p1 and p1 points to y, therefore p2 points to y.

If we desire this behavior, we can declare p2 to be a pointer to a pointer and derefrence twice.

```
//Pointer to a Pointer: Changing p1 changes p2
int main(void)
{
  int x=1, y=2, *p1, **p2;    //p2 is a pointer to a pointer
  p1=&x;
  p2=&p1;
  printf("%d\t%d\n",*p1,**p2);
  p1=&y;
  printf("%d\t%d\n",*p1,**p2);
}
>>> 1   1
>>> 2   2
```

Thus changing p1 to point to y changes both p1 and p2. Note the contrast between this behavior and that of shared references, where changing one pointer does not affect the other. Pictorially, dereferencing p2 twice via **p2 means we must follow two sets of arrows to get the value of p2. We show the situation before and after the assignment p1=&y; in figured

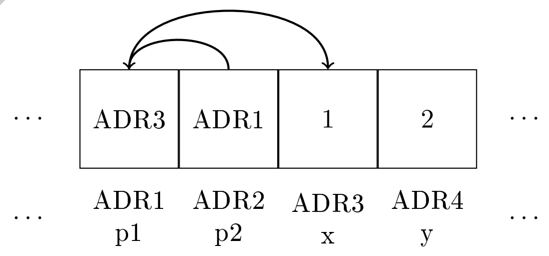

FIGURE 4.1.7. Pointers to Pointers - Before p1=&y;

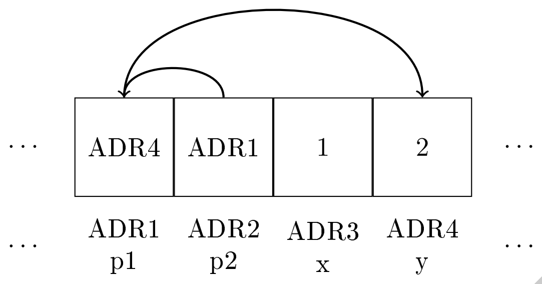

FIGURE 4.1.8. Pointers to Pointers - After p1=&y;

4.1.3. Call by Reference.

4.1.4. Call by Value.

4.2. Python Style Variables

Variables in Python are are much simpler than in C. In python, there is no variable declaration or initialization, only assignment. It is impossible for a variable to exist without a value. Python does not allow us to manipulate pointers directly. However, since the canonical implementation (i.e. interpreter) of Python is written in *C*, pointers underlie much of the behavior that may be strange or confusing. Thus it is essential to understand the nature of points to understand Python.

You don't create variables to hold a custom type. You don't create variables to hold any particular type. You don't "create" variables at all. You give names to objects.

CHAPTER 5

Python Data Types/Structures

Programming is all about storing and manipulating data. In order to accomplish these tasks effectively, it helps both the programmer and the program to know some basic information about the data, ideally without even having to examine it. For this reason, most programming languages will categorize data into **types**. Types serve two main purposes:

- Types provide implicit context for many operations. In C, for instance, the expression `a + b` will use integer addition if `a` and `b` are of integer type; it will use floating-point addition if `a` and `b` are of double (floating-point) type.
- Types limit the set of operations that may be performed in a semantically valid program. They prevent the programmer from adding a character and a record, for example, or from taking the arctangent of a set, or passing a file as a parameter to a subroutine that expects an integer.

DEFINITION 35. Type checking, type clash, strongly typed, statically typed, dynamically typed.

Type checking is the process of ensuring that a program obeys the language's type compatibility rules. A violation of the rules is known as a type **clash**. A language is said to be **strongly typed** if it prohibits, in a way that the language implementation can enforce, the application of any operation to any object that is not intended to support that operation. A language which is not strongly typed is said to be weakly typed. A language is said to be **statically typed** if most type checking can be performed at compile time, and the rest can be performed at run time. A language which is not statically typed - i.e. one in which most type checking is performed at run time is said to be **dynamically typed**.[**16**, 289] For more on the ontological nature of types, see [**16**, 293].

5.1. Mutable Types

5.1.1. Lists. On the surface, a list is simply a Python implementation of a sequence. The syntax for the nth element of a list is simple:

```
>>> L=[1,2,3]    #create a new list
#Lists start counting at 0
>>> L[0]                  #1
#Negative index counts backward from last element
>>> L[-2]                 #2
```

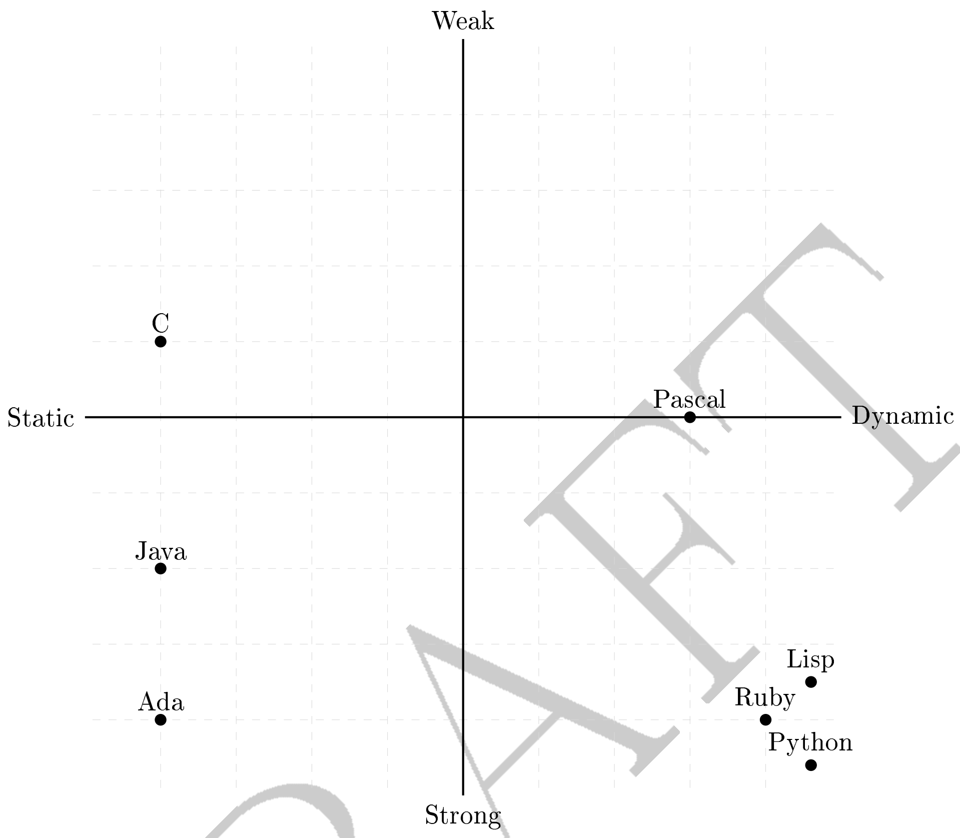

FIGURE 5.0.1. The Static/Dynamic - Strong/Weak Typing Plane

In languages such as C, the type of elements the list contains must be declared before the list is created. This makes it easy to say a certain list L is a sequence on (i.e. whose codomain is) strings or integers.

Lists can exhibit some interesting behavior

```
>>> L=[1,2,3]
```

5.1.2. Dicts. A **dictionary** is basically a sequence, except the elements of the index set no longer have to be natural numbers. Indeed the elements of the index set, called **keys** in this context, can be objects of any immutable type.

5.2. Immutable Types

5.2.1. Strings.

5.2.2. Numbers.

5.2.3. Tuples.

(1) https://docs.python.org/3/tutorial/datastructures.html
(2) https://docs.python.org/3/library/stdtypes.html

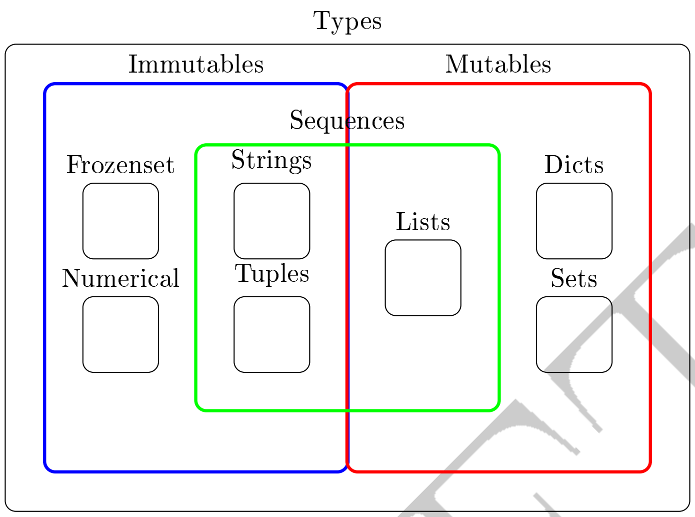

FIGURE 5.2.1. Python Type Hierarchy

5.3. Functions

5.3.1. Declaration.

5.3.2. Arguments.

5.3.2.1. *Argument Passing Methods.*

DEFINITION 36. Argument Passing
The way that objects are sent to functions as inputs.

Argument Passing Methods include:

(1) Pass by Value,
(2) Pass by Refrence (C)
(3) Pass by Assignment (Python)

5.3.3. Declaration.

5.3.4. Returning Values.

CHAPTER 6

Scopes and Namespaces

(1) Namespaces are just mappings from names to objects, stored as dictionaries
(2) `locals()` and `globals()` functions show these dictionaries for a given name space
(3) We can actually assign new variables by editing the dictionaries directly: `globals()['mynewkey'] = 'mynewvalue'` creates a new name binding in the global scope. Entering mynewkey at the prompt returns `'mynewvalue'`, just as if we had defined `mynewkey='mynewvalue'`

Definition 37. Python uses **Lexical Scoping**; variable scopes are determined entirely by their locations in the source code of your program files, not by function calls.

Definition 38. LEGB Lookup
Python looks up names using the **LEGB method**, which searches the local, enclosing, global, and built-in scope, stopping when it finds the first match. For example, the following code effectively overwrites the built-in `print()` function by assigning `A` to the name `print` in the global scope. Since Python uses LEGB lookup, it finds the user definition of `print=A` in the global scope before the builtin `print()` which lives in the builtins module. Thus typing `print` returns `A`. In order to get the regular print function back, we have to import it from the builtins module.

```
>>> print
<built-in function print>
>>> print="A"
>>> print
"A"
>>> from builtins import print
```

Exercise 39. Overwrite the built-in `print()`function with a function of your own. Your function should accept all possible arguments (although you can ignore all but the first argument) and use the built-in `print()` function to return a *modified* output to the user.

```
def print(text, *args):
        builtins.print("<<< {} >>>".format(text))
```

CHAPTER 7

Flow Control

7.1. Loops

7.1.1. For Loops.

7.1.2. While Loops.

7.1.3. ADVANCED: List comprehensions: Very powerful tool for building lists, based on the "set builder" notation of math
Reference: https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions

7.2. Iterators

7.3. Generators

7.4. Conditionals

7.4.1. If, Elif and Else. Do not confuse the logical conditional if statement with that used in programing.

```
>>> if x < 0:
...     print('Negative')
... elif x == 0:
...     print('Zero')
... else:
...     print('Positive')
...
```

(1) Reference: https://docs.python.org/3/tutorial/controlflow.html
(2) Libraries
    (a) Q: what is a library?
        A: a collection of functions
    (b) Common Libraries
        (i) urllib2, Numpy, Scipy, BeautifulSoup, matplotlib.
(3) Boolean Logic
    (a) and, or, not
    (b) Reference: https://docs.python.org/3/library/stdtypes.html

7.5. Operators

7.5.1. Basic Operators:

(1) $+, -, *, /,$
(2) Assignment operator: $=$

(3) Exponent: $**$
(4) Remainder: %

7.5.2. Comparison operators.

(1) Greater than: $>$, Less than $<$,
(2) Greater than or equal to: $>=$, Less than or equal to: $=<$,
(3) Test for equality $==$
(4) Full list of operators with explanations:
http://www.tutorialspoint.com/python/python_basic_operators.htm
https://docs.python.org/3/library/stdtypes.html
Q: How do I ask python for a full list of operators?

```
>>>import operator
>>>help(operator)
```

CHAPTER 8

User Interaction

8.1. Input and Raw Input

TODO

CHAPTER 9

Object Oriented Programming

Over time, different schools of thought have developed based on different ways of viewing programming. These schools of thought are called programming "paradigms."[**?**, xvi] One paradigm which has become extremley popular is known as **object oriented programming** (**OOP**) in which each component of a software program is an "object" which contains both data and behavior. When done properly, OOP provides a powerful level of abstraction above the computer hardware.

In Python, everything is an object; even numbers and strings! This means that unlike languages such as C and Java, Python has no primitive types. Thus everyone who has programmed in Python has some experience with objects. However, merely using objects does not make our code as being object oriented. To qualify as being truly object oriented, the design of our programs must be purposefully designed to leverage the features and structures of OOP. The first, and most basic structure of OOP is the class.

DEFINITION 40. A **class** is a coding structure and device used to implement new kinds of objects in Python that support inheritance. In Python, classes are created with the class statement. [**11**, 783]

Classes

```
>>> class myname:          #An empty class
...      pass              #Every class must contain some statement
```

are designed to create and manage new objects and support inheritance -a mechanism of code customization and reuse.

Benefits of classes:

- Multiple Instances
- Customization by Inheritance
- Operator Overloading

At its base, OOP is two things, a special first argument in functions (to receive the subject of a call) and inheritance attribute search to support programming by customization.

9.0.1. Objects and Classes.

DEFINITION 41. Classes and Instances

Classes: serve as instance factories. Their attributes provide behavior - data and functions - that is inherited by all the instances generated from them

Instances: represent the concrete items in a program's domain. Their attributes record data that varies per specific object

Over time, different schools of thought have developed,
so a paradigm is a way of viewing programming

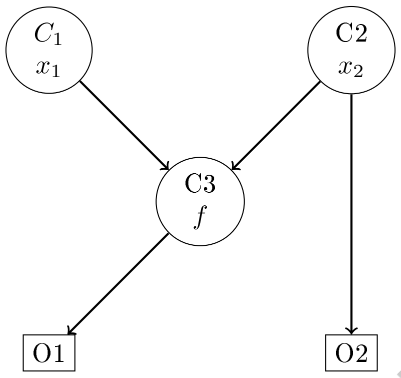

FIGURE 9.0.1. An OOP Inheritance Hierarchy

9.1. Turning Words into Code

For me this is a 4 step process:

(1) Understand the Problem: translate it into your own words
(2) Break the problem up into logically contained steps
(3) Pseudo-code these steps (this is less necessary in python)
(4) Write the python code

9.2. Example: Calculating Averages

Problem: write a function *list_mean* in python which computes and returns the average of a list of numbers. Use no built in functions besides `len(x)`.

(1) Understand the Problem: translate it into your own words
    (a) Recall that the average of a vector (or tuple) of numbers $(x_1, x_2, \dots x_n)$ is given by:

$$\frac{1}{n}\sum_{i=1}^{n} x_i$$

    This is the quantity which we're supposed to compute
(2) Break the problem up into logically contained steps
    (a) First we need to declare our function *list_mean*
    (b) How do we actually compute $\frac{1}{n}\sum_{i=1}^{n} x_i$?
        (i) We declare a variable *list_sum* and set it to 0
        (ii) Then we iterate through the list, adding each element to *list_sum*.
        (iii) We then divide *list_sum* by $n = len(list)$. This is the mean, as per the formula.
    (c) Finally, we return the mean we have calculated
(3) Pseudocode these steps
    (a) The python language is so similar to pseudocode, that I'm going to skip it.
(4) Write the python code

```
def list_mean(list):
        list_sum=0
        for element in list:
            list_sum = list_sum+element
    mean=list_sum/len(list)
    return(mean)
```

One thing to note here: in *Python 2*, when x, y are integers, python uses integer division, which is defined as:

$$x/y = \lfloor x/y \rfloor$$

where $\lfloor x/y \rfloor$ is called the *floor function*, and is defined as being equal to the largest integer in $\mathbb{Z} = \{\cdots -3, -2, -1, 0, 1, 2, 3, \cdots\}$ less than or equal to $\lfloor x/y \rfloor$. Thus in python 2.7:

```
>>>5/2=floor(2.5)=2
```

However, as of *Python 3* this has been changed, and the default division is now floating point division, which is basically the division we're familiar with, but with limited precision. So, this calculation will be rounded down in *python 2,* but not in *python 3.*

CHAPTER 10

Troubleshooting

10.1. Built-in *help* Function.

>>> $help(x)$, where x is an object of the type you're interested in. For instance:

```
>>>x=list()
>>>help(x)
```

will bring up help for lists.

10.2. Documentation Maintained by the Python Software Foundation

10.2.1. Best Practices.

(1) https://www.python.org/dev/peps/pep-0008/

10.2.2. Data Structures.

(1) https://docs.python.org/3/tutorial/datastructures.html

10.2.3. Data Types/Hierarchy.

(1) https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy

10.2.4. Built-in Functions:

(1) https://docs.python.org/3/library/functions.html#zip

10.3. Stack Overflow

Stack Overflow is a question and answer site for programmers.
http://stackoverflow.com/

10.4. Computerphile

Computerphile is a Youtube channel discussing interesting topics in computer science. Videos are presented in a technical, but accessible way. This is a great way to learn about computer science history, trendy topics (such as the iPhone 1/1/1970 hack), as well as material you see in class.
https://www.youtube.com/channel/UC9-y-6csu5WGm29I7JiwpnA

10.5. What's the Difference Between a List and a Tuple?

Answer: To fully answer this question, we need to look at the Python Standard Type Hierarchy, documented in the first reference below. This tells us how different types are related. Both lists and tuples are of the type family known as "Sequences". Sequences "**represent finite ordered sets indexed by non-negative numbers**. The built-in function $len()$ returns the number of items of a sequence. When the length of a sequence is n, the index set contains the numbers $0, 1, ..., n-1$. Item i of sequence a is selected by $a[i]$. Sequences are distinguished according to their mutability. An object of an immutable sequence type cannot change once it is created."

(1) Lists: have the following properties
  (a) Lists are mutables.
(2) Tuples have the following properties
  (a) Tuples are immutable: this means they are unable to be changed after they are created. Try this in python:

```
>>>tuple=(5,3)
>>>tuple[1]
3
>>>tuple[1]=7
error
```

  Clearly then, tuples are immutable, we cannot reassign *tuple[1]* to be 7.
  (b) They use less memory and iterate faster than lists. Use a tuple when you don't need to edit values later. This only really matters when working with large data.
  (c) References:
  https://docs.python.org/3/reference/datamodel.html#the-standard-type-hierarchy
  https://docs.python.org/3/glossary.html

10.6. Why are Best Practices Important?

Answer: Sometimes, you may encounter a situation where the "right answer" is a matter of perspective. In this case, check if there are best practices to which you can adhere. They also help you're code be readable to others and look professional. This is critical if you intend to get a job programming. See the list of best practices here:
https://www.python.org/dev/peps/pep-0008/

10.7. How do I Assign Multiple Variables on One Line?

Answer: The following code sets $a = 1, b = 2, c = 3$. It can be extended for arbitrarily many times >>> $a, b, c = 1, 2, 3$.

Part 3

Configuring a Development Environment

Your development environment is a place for your code to live and for you to work. Properly configuring this environment is essential for programmer productivity and happiness. In this section, we will examine several components which together constitute a development environment. Some of the topics we will discuss, like the operating system, file system, and version control are quite complex and merit extensive study in their own right. Our goal is merely that you be familiar enough with these topics that they do not impede your programming efforts. A detailed bibliography is provided for interested readers.

10.8. The Operating System

"An **operating system** (OS) is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware". [**19**, 3] Services that most operating systems provide include storage management (namely a file system and an interface for programs to interact with that file system), memory management, and CPU scheduling[1]. Different operating systems may provide these basic services in fundamentally different ways. Because of these differences, programs and tools which run on one operating system may not run on another. We say that such programs are **non-portable** or that they are are **operating system specific**. Since your operating system may limit which programs you can run, your choice of OS is is a key factor in your development environment.

Examples of operating systems include Microsoft Windows, macOS, Android, Linux, Unix. The study and design of operating systems are important topics in computer science. Interested readers may consult [**19**] for more information.

10.8.1. Virtual Machines. The fundamental idea behind a **virtual machine** (**VM**) is to "abstract the hardware of a single computer (the CPU, memory, disk drives, network interface cards, and so forth) into several different execution environments, thereby creating the illusion that each separate environment is running on its own private computer" [**19**, 711]. In other words, VMs allow (or give the appearance of allowing) several different operating systems to run on the same hardware, at the same time! The "underlying hardware system that runs the virtual machines" is known as the **host system**, and the virtual machines are known as **guest systems**. VMs are created and run by a program known as a **virtual machine manager** (**VMM**) (or **hypervisor**) [**19**, 712]. It may surprise the reader to learn that VMs are not a new idea. The first commercial VMs were implemented by the IBM VM/370 operating system in 1972 for use on its System/370 mainframes. The descendants of this operating system are still commercially available in 2018 [**19**, 713]!

10.8.1.1. *Benefits of Virtual Machines.*

Flexibility. Virtual machines allow several different operating systems to run on the same hardware, at the same time! This means you can run Windows or macOS as your everyday operating system and have access to all your familiar tools and programs, but also use Linux programs through your VM. The flexibility of VM is great for experimentation and study; a student with a windows PC can try out any

[1]Modern computers have few processors but run many programs. CPU scheduling is the art/science of deciding which programs run when, or in other words, how to divide the CPU's resources.

number of operating systems including several versions of Linux, Solaris, Windows, macOS, BSD/Unix and more. This can be done for free, without changing any hardware. Whats more, an entire virtual machine may be saved, copied, transferred to another computer like a regular file, and then restarted!

Security. Virtual machines can protect both the host system and other VMs from both accidental and intentional damage. We quote [**19**, 714]:

> One important advantage of virtualization is that the host system is protected from the virtual machines, just as the virtual machines are protected from each other. A virus inside a guest operating system might damage that operating system but is unlikely to affect the host or the other guests. Because each virtual machine is almost completely isolated from all other virtual machines, there are almost no protection problems

Additionally, a VM can protect the host system and other VMs from user error. For instance, entering the wrong command on the host machine could destroy an entire file system, obliterating important documents, photos, as well as all virtual machines running on the host system. However, entering the wrong command on a virtual machine will at worst destroy that virtual machine, which can then be quickly rebuilt using some of the tools we will discuss later.

Development/Production Parity. Suppose you are working on a project with several other developers. This project might have many dependencies, i.e. programs and tools that the project requires in order to function. Thus before a developer can even *start* to work on the project, he or she must spend time installing all the dependencies. Since *every* developer must repeat this process, the time wasted is substantial. Even if all developers have installed the correct dependencies, they may configure them differently (they may have different paths, folder structures, etc). This leads to nasty "**works on my machine**" or "**portability bugs**", where code can cause errors on some developer's computers but not others. This includes the case where a program may run error free on a developer's machine (say their personal laptop), but not on the production machine (usually a large server) which will actually run the program.

Developers may overcome these problems via a process known as **templating**, in which an administrator creates a standard virtual machine image, including an installed and configured guest operating system and applications. Users may then **clone** (i.e. copy) this virtual machine and boot it on their host machines. Thus each developer obtains an identical virtual machine, and portability bugs are eliminated.

10.8.1.2. *Costs.* Virtual machines consume system resources, especially memory, processor time, and storage. This means running a VM may negatively impact the performance of an older PC. Additionally, there is an upfront cost to learning about, setting up, and configuring a virtual machine. This process can be confusing and time consuming.

10.8.1.3. *Setting up a Virtual Machine.*

A number of tools exist to make setting up and configuring a virtual machine as easy as possible. We recommend using Vagrant together with VritualBox. Vagrant is a free "tool for building complete development environments, sandboxed in a virtual machine. Vagrant is cross platform and "behaves identically across Linux, Mac OS X" [**7**, 1,6]. VritualBox is a free, open source, and cross-platform virtualization software (hypervisor) backed by Oracle. It is available for Linux, Mac OS

X, Windows" [**7**, 7]. Getting a basic VM running with Vagrant and VritualBox may take as little as five minutes. Definitive resources on Vagrant and VritualBox may be found here: [**7**], [**13**].

10.8.1.4. *Wisdom of the Crowd.* Now that we have learned a bit about operating systems, we might be interested in what OS professional developers actually prefer? The 2018 Stack Overflow survery of developers provides enormous insight into current trends in the software industry. Out of the 71,222 Response, 98% of developers used either Windows, macOS, or Linux as their primary operating system. We should note however, that the use of virtual machines confuses this question somewhat.

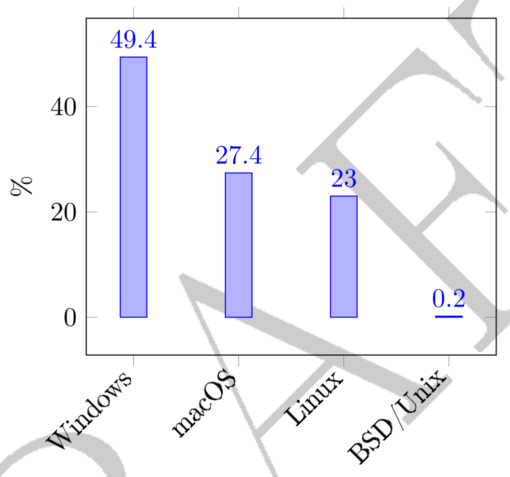

FIGURE 10.8.1. Developers' Primary Operating Systems, 2018, 71,222 Responses

10.8.1.5. *Windows Subsystem for Linux.* As of Windows 10 build number TODO, Windows 10 provides the option for the user to install a compatibility layer known as the Windows Subsystem for Linux (WSL). In this subsystem, windows files are given extra properties which enable them to interact with Linux style files. This gives the user the ability to run Linux apps nativley like a virtual machine.

The WSL has one big advantage over a virtual machine - it is much faster (source)

Like a VM, the WSL can be quicky torn down and rebuilt.

Unlike a VM, it is possible to enter a simple terminal command and destory your system

I haven't explored provisioning the WSL TODO

10.9. File System

Of all the services the operating system provides, the file system is the one we interact with the most. The **file system**

> provides the mechanism for storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection

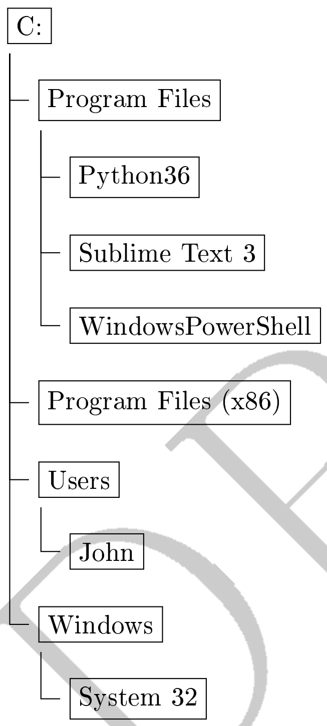

FIGURE 10.9.1. Windows

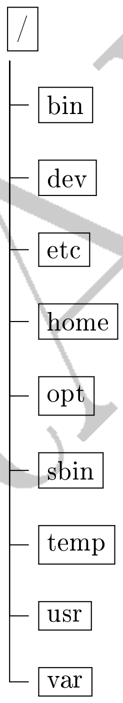

FIGURE 10.9.2. Linux

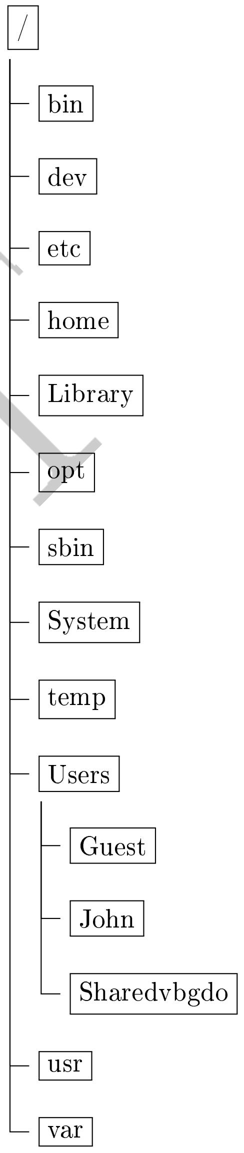

FIGURE 10.9.3. macOS

FIGURE 10.9.4. Directory Structures of Common File Systems

of **files**, each storing related data, and a **directory structure**, which organizes and provides information about all the files in the system[**19**, 503]

The simplest place to store a file is on your computer's hard drive. This is known as **local storage**. Alternativley, you may store your programs remotley by using the internet to transfer your files to and from a computer (or network of computers) known as a **server**. This is known as **remote**, **networked**, or **cloud**

storage.[2] Regardless whether we store our programs in remote or local storage, we need to understand the way this storage is organized - that is, we must understand the computer's file system.

Since we will write many programs, we need a way of organizing them. We start by putting them all in one place. To do this we make a folder - a file which may contain other files, called `development`. Now that our files are all collected, we can organize them by creating sub-folders in If you program in multiple languages, it may be useful to create one sub-folder for each language. Thus a beginning programmer might have folder system that looks like this. Large projects should be saved together in one folder.

In normal use, each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process. when reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user must manually either specify a math name or change the current directory to be the directory holding that file. - TO CITE - OS

We illustrate such a file system below. Folders are $\boxed{\text{boxed}}$ to distinguish them from files, which all have an extension signifying their program association.

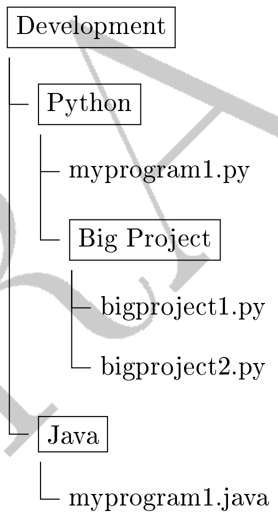

FIGURE 10.9.5. File System

In the next section, we illustrate a tool for navigating your file system, and much more; the system shell.

10.10. The Shell

One of the primary ways a user can interact with their operating system is via a special program known as a **command line interpreter (CLI)** or **shell**. The shell gives users a powerful text based interface with the OS. Typically, one

[2]Economical cloud storage is offered by Google's Google Drive, Apple's iCloud, Microsoft's oneDrive, and Dropbox. As of July 2018, these services offer 15, 5, 5, and 2 GB of free storage respective, and offer paid options for more storage.

programs the shell using a scripting language such as Bash. The Bash commands are then translated into a form the operating system understands by the Bash CLI (or Bash shell). The user input to and machine output from the shell are displayed by a **terminal** or more precisely, a **terminal emulator**. Since a scripting language and its interpreter are nearly always used together, one name is frequently used to refer to them both.[3]

This may sound confusing, but the situation is not that different in Python. The Python programming language is related to, yet distinct from the Python interpreter, however people often abuse terminology and speak of "starting Python," "running Python," or "getting an error from Python" when in fact, they it is the Python interpreter which is performing these actions. A language is just a set of symbols and rules, it is the interpreter (or compiler) which performs the actions.

Things are even worse on Windows. PowerShell is actually one name for three different things; a programming language, a program which interprets the PowerShell language (CLI), and the terminal which handles the I/O! Luckily, because we now understand the meaning of these terms, namely that the terminal is distinct from the CLI, we know we should be able to swap out the default PowerShell terminal for a superior one! Indeed, there are many free Windows terminal emulators available including Hyper, ConEmu, and cmder. The ready may experiment with these at leisure to find a suitable alternative.

When PowerShell is paired with a modern terminal emulator, it is entirely usable. However, it is not without its some drawbacks. Firstly, it only works on Windows. Secondly, Bash has been available since 1989 and Linux and Unix (*nix) developers have become accustomed to it. In some ways, Bash has become *the* canonical shell. This is problematic for Windows developers, as Windows doesn't ship with a Bash shell. Fortunately, there are tools such as **Cygwin** which provide the feel of a Bash shell in Windows! The Windows Subsystem for Linux also provides a Bash shell. The hardcore Bash enthusiast can handle nearly tasks from the WSL Bash shell.

In normal use, each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process. when reference is made to a file, the current directory is searched. If a file is needed that is not in the current directory, then the user must manually either specify a math name or change the current directory to be the directory holding that file.

The shell is a great way to explore or **traverse** the previously described file system. Before, we get started learning some shell commands, we need some words of caution.

The shell is a powerful tool, which reaches deep into your computer. Entering the wrong command can have serious repercussions, including the deletion of all your data. **Never run code from untrusted sources in your terminal** (unless you are using a disposable virtual machine)!

Thankfully, the basic commands for this are the same in both Bash and PowerShell

[3]The Bash programming language is interpreted by the Bash shell, which displays its output in the Terminal application. In other words, Bash is an interpreter for an interpreted language... called Bash.

```
$ pwd   #print the present working directory
$ ls    #print a list of files in the current directory
```

10.11. The Text Editor/Integrated Development Environment (IDE)

At a basic level, a computer program is like a book or essay; it is simply lines on a page. Just as a writer uses a word processor, a computer programmer uses either a **text editor** or **integrated development environment** (**IDE**) to put words to the page. Modern IDEs provide a wealth of features to the developer, including the following:

- Error checking,
- Code navigation
- Code completion
- Code coloring/syntax highlighting
- Version control integration
- Integrated debugging tools
- Extensions/Add-ons/Plugins

A classic example of an IDE is IntelliJ IDEA, a Java IDE which also forms the foundation for the popular PyCharm Python IDE.

Text editors are generally simpler, smaller, and faster than IDEs, but may still have some of the features of an IDE. For example, Sublime Text 3 is an excellent text editor which incorporates code navigation, syntax highlighting, and plugin support. Recently, programs such as VSCode are have blurred the line between text editor and IDE.

10.12. Version Control

Suppose you are web developer getting ready to release/deploy a new update to your website. You're excited to share your work with the world, but then... disaster! When the new code goes live, it causes your website to crash (perhaps you didn't use a virtual machine, and have a portability bug)! Your boss demands you fix it immediately. You've changed hundreds of lines of code since the old, working version of the website. You can try to remember which lines they are and delete them one by one, but this can introduce new bugs if you "misremember" *any* of them. If only you had saved a copy of the previously working code.... This is where version control comes in. Each time a change is made to your project, an exact accounting is made of the changes, including the time, the author, and a description of the change. That way, should a disaster occur, you can easily rollback the changes to a time when you knew the code works. Once this is done, you can troubleshoot at your leisure, and find that pesky bug.

Formally, a **version control system** (**VCS**) is tool that manages and tracks different versions of software or other content.[4]. The goal of a VCS is to "develop and maintain a repository of content, provide access to historical editions of each datum, and record all changes in a log." [**10**, 1]. Today, Git is by far the most common VCS, with 87% of software developers reporting they used Git.[**20**] Git was created

[4]A VCS may also be referred to as a **source code manager** (**SCM**) or **revision control system** (**RCS**)

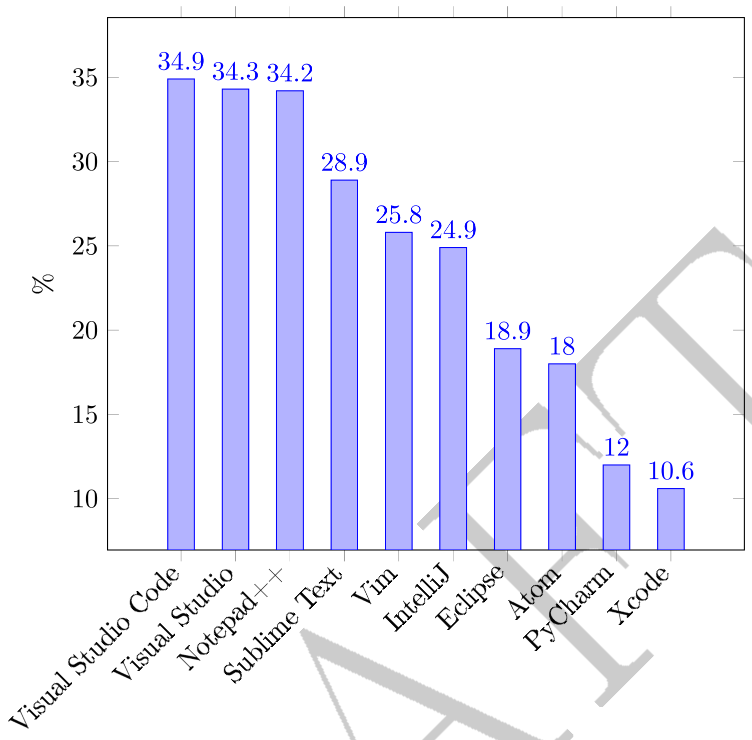

FIGURE 10.11.1. Most Popular Development Environments, 2018, 75,398 Respondents

in 2005 by Linus Torvalds, the inventor of Linux, to support the development of the Linux Kernel.[**10**, 1,5].

As a new programmer, using version control is probably overkill. In general the need for version control increases with program complexity, and the number of developers. However, you should still know it exists.

Detailed information on Git can be found in [**10**] or in [**17**] which is available for free under a Creative Commons license here[5].

[5]https://git-scm.com/book/en/v2

Index

Bibliography

[1] Bartle, R. G., and Sherbert, D. R. *Introduction to Real Analysis*. Wiley, 2011.

[2] Brian W. Kernighan, D. R. *The C Programming Language*. Microsoft Press, 1988.

[3] Cain, G. L. *Introduction to General Topology*. Addison-Wesley, New York, 1994.

[4] Corsair. Corsair vengeance lpx 16gb (2x8gb) ddr4 dram 2400mhz c16 desktop memory kit - black (cmk16gx4m2a2400c16), 2018.

[5] David A. Patterson, J. L. H. *Computer Organization and Design*. Elsevier LTD, Oxford, 2013.

[6] Digi-Key Electonics. Issi, integrated silicon solution inc is61wv51216edbll-10tli, 2018.

[7] Hashimoto, M. *Vagrant: Up and Running*. O'Reilly UK Ltd., 2013.

[8] IBM. Ibm 38l7302 lto7 ultrium7 15tb rw data cartridge (new), 2018.

[9] International Organization for Standardization. Quantities and units–part 2: Mathematical signs and symbols to be used in the natural sciences and technology. techreport ISO 80000-2:2009, International Organization for Standardization, Geneva, Switzerland, Dec. 2009.

[10] Loeliger, J. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, 2009.

[11] Lutz, M. *Learning Python*. O'Reilly UK Ltd., 2013.

[12] Miller, G. A. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review 63*, 2 (1956), 81.

[13] Oracle. *Oracle VM VirtualBox*, 5.1.26 ed. Oracle Corporation, Redwood City, CA, 2018. https://download.virtualbox.org/virtualbox/UserManual.pdf.

[14] Rosen, K. H. *Discrete Mathematics and Its Applicaitons*, 7 ed. McGraw Hill, New York, 2012.

[15] Samsung. Samsung 850 evo 500gb 2.5-inch sata iii internal ssd (mz-75e500b/am), 2018.

[16] Scott, M. L. *Programming Language Pragmatics, Third Edition*. Morgan Kaufmann, 2009.

[17] Scott Chacon, B. S. *Pro Git*. APRESS L.P., 2014. https://git-scm.com/book/en/v2.

[18] Seagate. Seagate bare drives 8tb barracuda sata 6gb/s 256mb cache 3.5-inch internal hard drive 3.5 internal bare/oem drive, 2018.

[19] Silberschatz, A., Galvin, P. B., and Gagne, G. *Operating System Concepts*. John WIiley & Sons INC, 2012.

[20] Stack Overflow. Developer survey results. Website, 2018. https://insights.stackoverflow.com/survey/2018/.

[21] Sundstrom, T. *Mathematical Reasoning Writing and Proof*, 1.1 ed. Pearson, New York, 2013.

APPENDIX A

ASCII Table

Dec	Hex	Binary	Char	Dec	Hex	Binary	Char	Dec	Hex	Binary	Char
32	20	00100000	Space	64	40	01000000	@	96	60	01100000	‘
33	21	00100001	!	65	41	01000001	A	97	61	01100001	a
34	22	00100010	"	66	42	01000010	B	98	62	01100010	b
35	23	00100011	#	67	43	01000011	C	99	63	01100011	c
36	24	00100100	$	68	44	01000100	D	100	64	01100100	d
37	25	00100101	%	69	45	01000101	E	101	65	01100101	e
38	26	00100110	&	70	46	01000110	F	102	66	01100110	f
39	27	00100111	’	71	47	01000111	G	103	67	01100111	g
40	28	00101000	(	72	48	01001000	H	104	68	01101000	h
41	29	00101001	)	73	49	01001001	I	105	69	01101001	i
42	2A	00101010	*	74	4A	01001010	J	106	6A	01101010	j
43	2B	00101011	+	75	4B	01001011	K	107	6B	01101011	k
44	2C	00101100	,	76	4C	01001100	L	108	6C	01101100	l
45	2D	00101101	-	77	4D	01001101	M	109	6D	01101101	m
46	2E	00101110	.	78	4E	01001110	N	110	6E	01101110	n
47	2F	00101111	/	79	4F	01001111	O	111	6F	01101111	o
48	30	00110000	0	80	50	01010000	P	112	70	01110000	p
49	31	00110001	1	81	51	01010001	Q	113	71	01110001	q
50	32	00110010	2	82	52	01010010	R	114	72	01110010	r
51	33	00110011	3	83	53	01010011	S	115	73	01110011	s
52	34	00110100	4	84	54	01010100	T	116	74	01110100	t
53	35	00110101	5	85	55	01010101	U	117	75	01110101	u
54	36	00110110	6	86	56	01010110	V	118	76	01110110	v
55	37	00110111	7	87	57	01010111	W	119	77	01110111	w
56	38	00111000	8	88	58	01011000	X	120	78	01111000	x
57	39	00111001	9	89	59	01011001	Y	121	79	01111001	y
58	3A	00111010	:	90	5A	01011010	Z	122	7A	01111010	z
59	3B	00111011	;	91	5B	01011011	[	123	7B	01111011	{
60	3C	00111100	<	92	5C	01011100	\	124	7C	01111100	|
61	3D	00111101	=	93	5D	01011101	]	125	7D	01111101	}
62	3E	00111110	>	94	5E	01011110	^	126	7E	01111110	~
63	3F	00111111	?	95	5F	01011111	_	127	7F	01111111	DEL

TABLE 1. The ASCII Table - Printable Characters Only