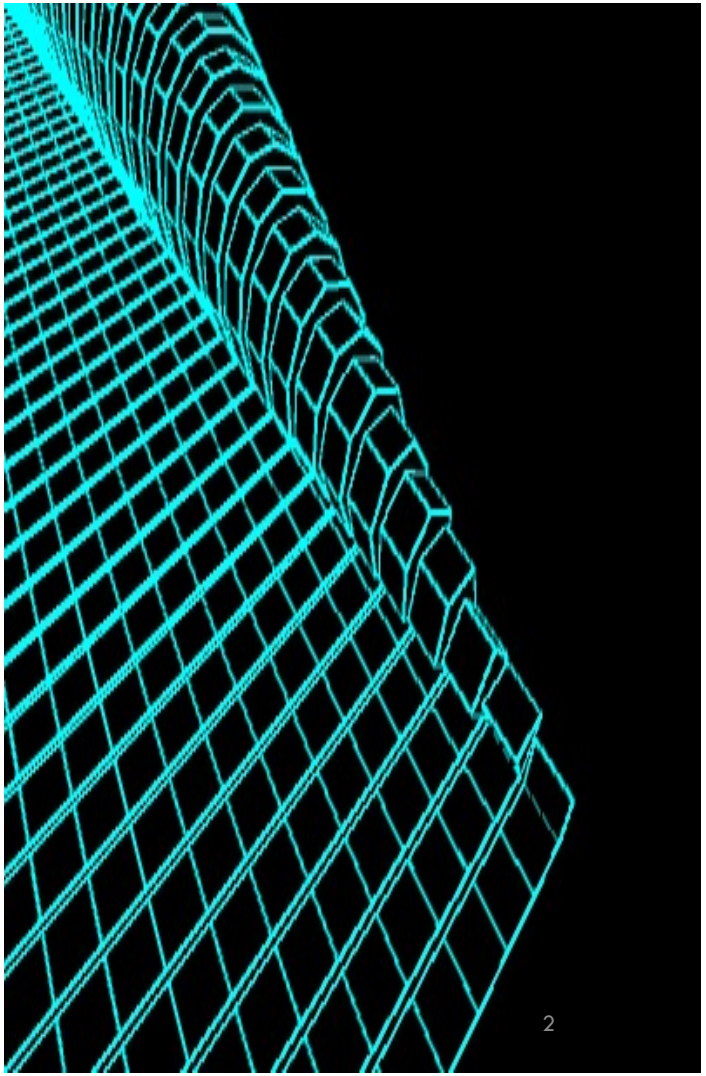
COMP 1510
Winter 2022

Programming Methods

Week 01: Introduction

Agenda for week 01

1. Introduction
2. Logistics
3. Communicating
4. Selected history of programming
5. Programming language levels
6. Programming language paradigms
7. Introduction to Python a high-level multi-paradigm language
8. Python keywords
9. The Zen of Python and the notion of programming philosophies and methods
10. The command line
11. Operators, operands, and expressions
12. Operator precedence
13. Commands like print()
14. Variables and assignment



INTRODUCTION

Hello! 👋


christhompsonmeets


CST_Chris


christhompsonteaches


christhompsonchats

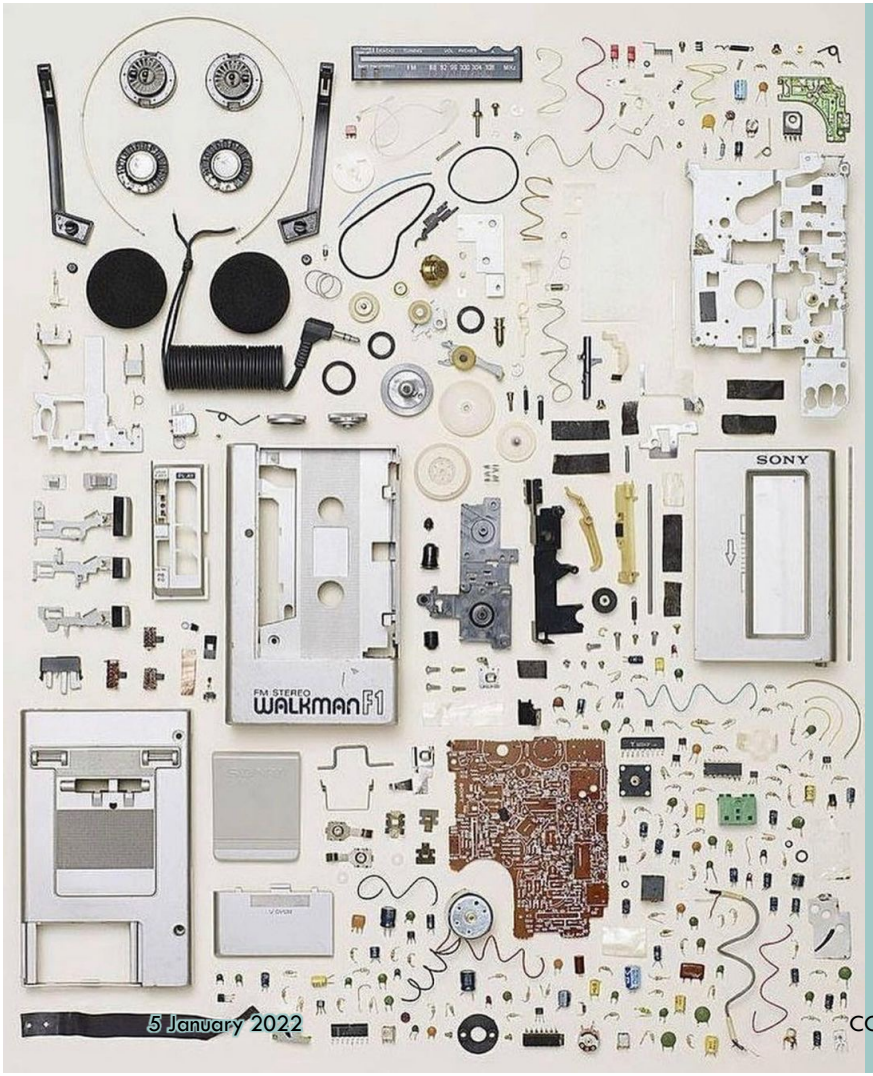

christhompsonquilts


chris_thompson@bcit.ca


bcit-comp1510-202210.slack.com

COMP 1510: Programming Methods

- Your **1st** programming course
- The ***most important*** course this term
- Prerequisite for:
  1. COMP 2800 (spring project course with me!)
  2. COMP 2522 (Object oriented programming with me!)
  3. COMP 2510 (Procedural programming using C with Seyed!)
- ***No matter what, come to class and lab and office hours and ask me lots of questions.***



When/where do we meet?

- Lectures with me:
  - Monday 11:30 – 12:20 (1 hour)
  - Thursday 3:30 – 5:20 (2 hours)
- Labs and tutorials with me too!
  - No, there are no labs/tutorials this week!
  - We will start next week!
- I will use Slack to communicate important information so stay logged in (download the app!)

What will you learn how to do?

- Design, implement, debug, and test simple object-oriented programs in a modern high-level programming language (Python!)
- Write programs using good software development processes including design before implementation, encapsulation, information hiding, message passing, decomposition, and testing
- Analyze and explain behaviour of simple programs involving fundamental programming constructs
- Apply the techniques of decomposition to break a program into smaller modules with well-defined interfaces
- Write automated tests to verify the correctness of your code
- Explain the representation and use of primitive data types and built-in data structures, including arrays and strings
- Explain the value of application programming interfaces (APIs) in software development
- Use variables, control statements, and input/output
- Use error handling to deal with exceptional circumstances
- And so much more!

LOGISTICS

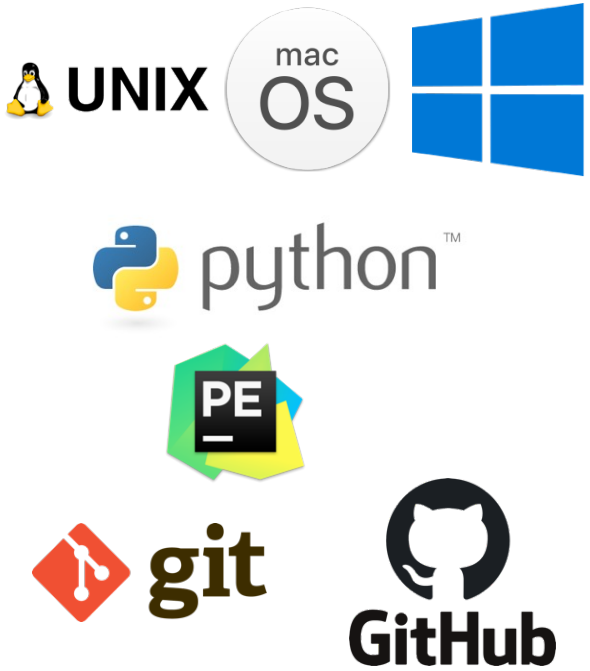
Evaluation criteria

10%	Weekly quizzes
10%	Attendance and participation
20%	Weekly labs
25%	6 independent short-term programming assignments
10%	Midterm assessment
25%	Final assessment

A passing grade is 50.0%

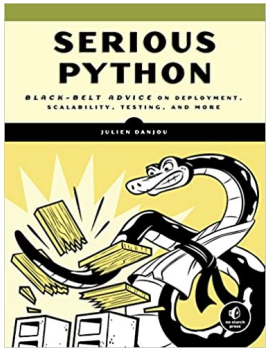
What resources do you need to get?

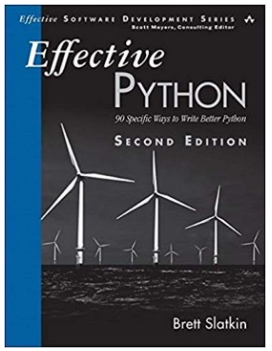
1. Windows, macOS, all flavours of *nix are fine
2. Python 3.10.1+ (more about that later)
3. PyCharm IDE (later…)
4. git and Github (later…)
5. And more (later…!)

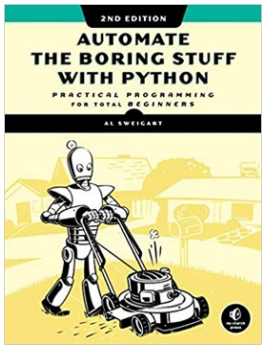


Optional (recommended) textbooks

1. Danjou, Julien. (2020). *Serious Python*. No Starch Press.
2. Ramalho, Luciano. (2015). *Fluent Python: Clear, Concise, and Effective Programming*. O'Reilly.
3. Slatkin, Brett. (2019). *Effective Python 2nd Edition*. Addison-Wesley.
4. Sweigart, Al. (2019). *Automate the Boring Stuff with Python 2nd Edition*. No Starch Press.







Where will course materials be available?

- BCIT Learning Hub > Desire to Learn (D2L) for slides, labs, assignments, links to online resources, grades
- I will post all official announcements on Slack
- Please use Slack for all communication, requests for help, etc.
- Please check Slack frequently. In fact, leave it on!
- I will use Calendly to schedule office hours (When is best? Discuss and set reps please advise me ASAP!)
- You will schedule your appointments with me at:

https://calendly.com/christhompsonmeets

Are there any rules about our work?

- You are encouraged to collaborate by:
  - Completing in-class and in-lab exercises in pairs when permitted
  - Helping each other understand material and assignments
  - Discussing requirements and approaches and debugging code together
- **Plagiarism** is not allowed:
  - Exchanging or sharing code snippets/solutions
  - Submitting someone else's work as your own (even part of it)
- Academic Integrity: www.bcit.ca/files/pdf/policies/5104.pdf

SLACK

So how do we communicate?

- With Slack!
- Download the client
- Our Slack workspace is called bcit-comp1510-202210.slack.com
- I'll post the invitation right now in the Zoom chat (it's also in the course News on the Learning Hub learn.bcit.ca)

Please sign up and send me a direct message in Slack today so I know you're in! Tell me your preferred first name, and what sort of computing experience you have!

HISTORY OF PROGRAMMING

A (brief) selected history of programming

Long long ago, in
a galaxy far, far away . . .

Let's select an arbitrary logical starting point:

•Electricity!

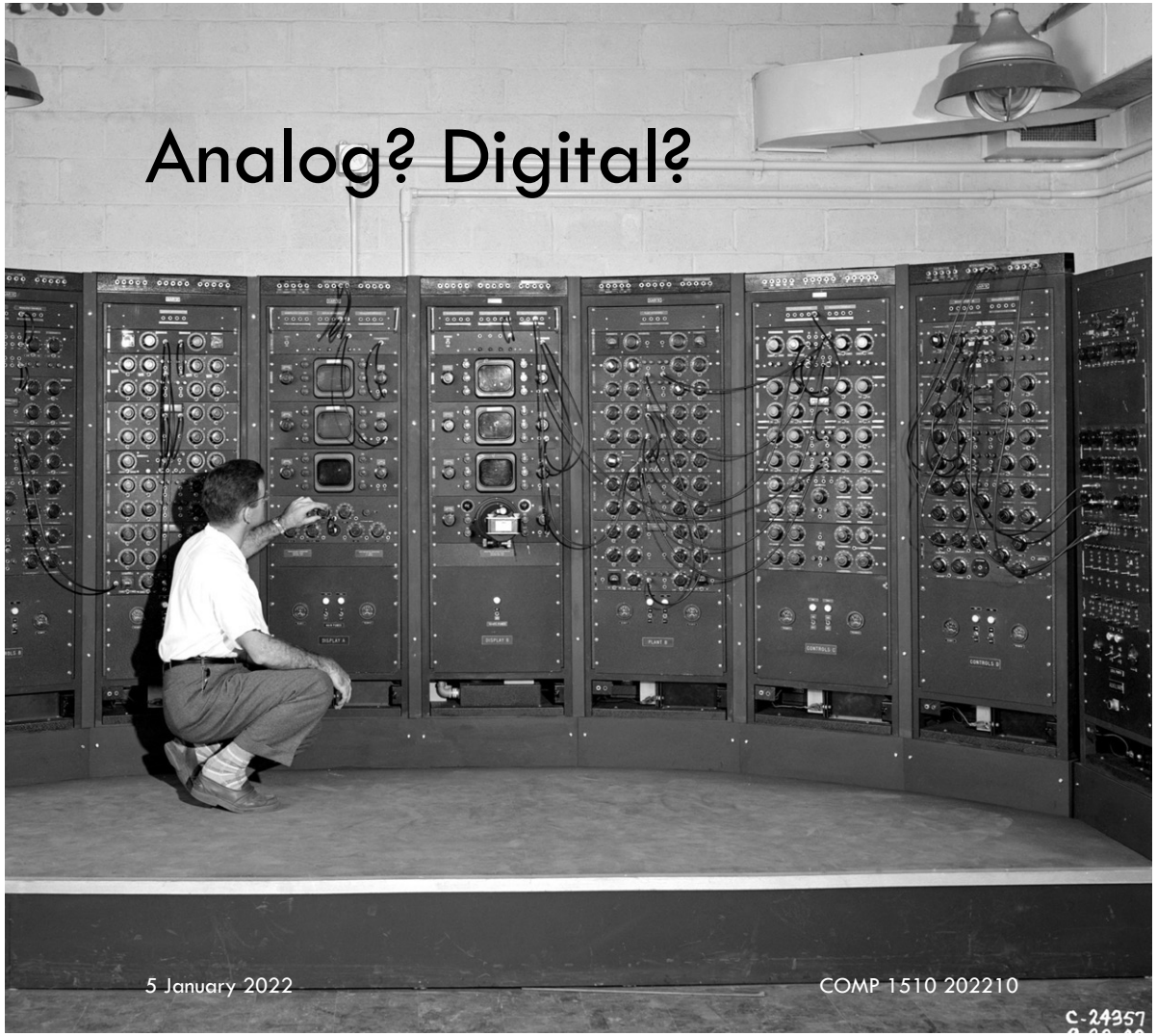
- In order to use it, we needed to control its flow
- We invented the **switch**
- **A switch controls the flow of electricity through a wire**
- At first, we did this **manually** (think Dr. Frankenstein)
- In the early 20th century, we learned how to control a switch electronically using…

Vacuum tubes!

What did we invent with vacuum tubes?

- Radio
- Television
- Radar
- Sound recording and reproduction
- Large telephone networks
- Industrial process control (think of how far THIS has evolved…!)
- **Analog and digital computers.**

Analog? Digital?



- **Analog:** signals are translated into electrical impulses of different amplitude (strength)
- **Digital:** information is translated into binary format (zero or one)

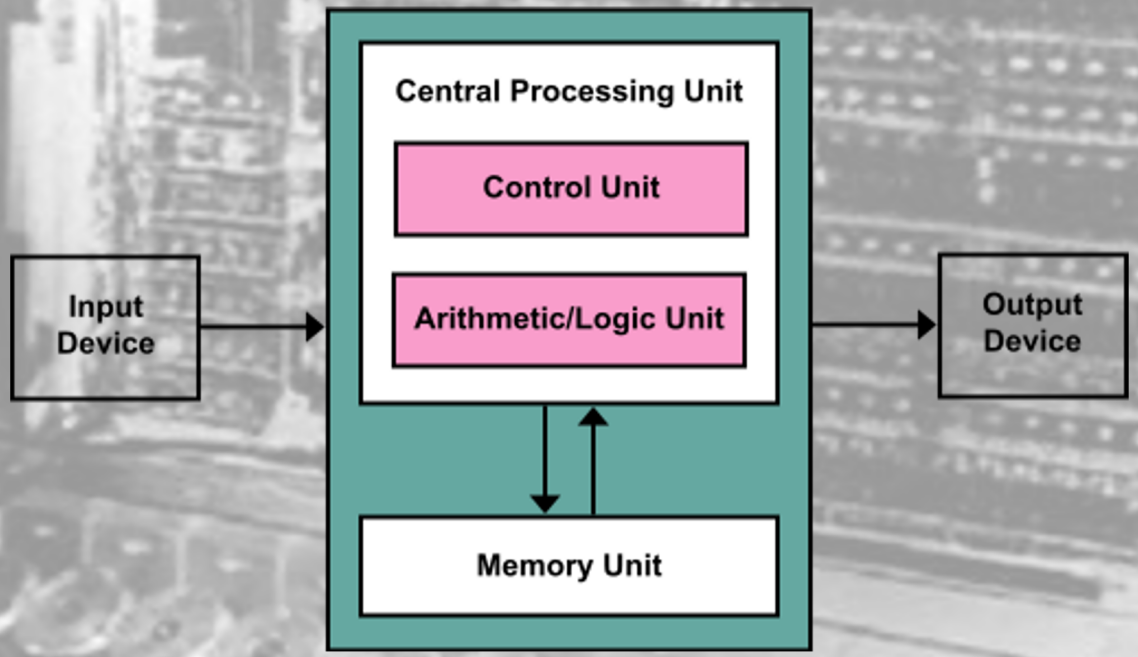
Analog computer
Lewis Flight Propulsion Lab 1949

1936: universal Turing machine

- Alan Turing described a set of
  1. Rules
  2. States
  3. Transitions
- Solves mathematical functions
- Determines whether a given word is in a language or not
- https://youtu.be/E3keLeMwfHY
- Used by John von Neumann in 1946 for his stored-program computer

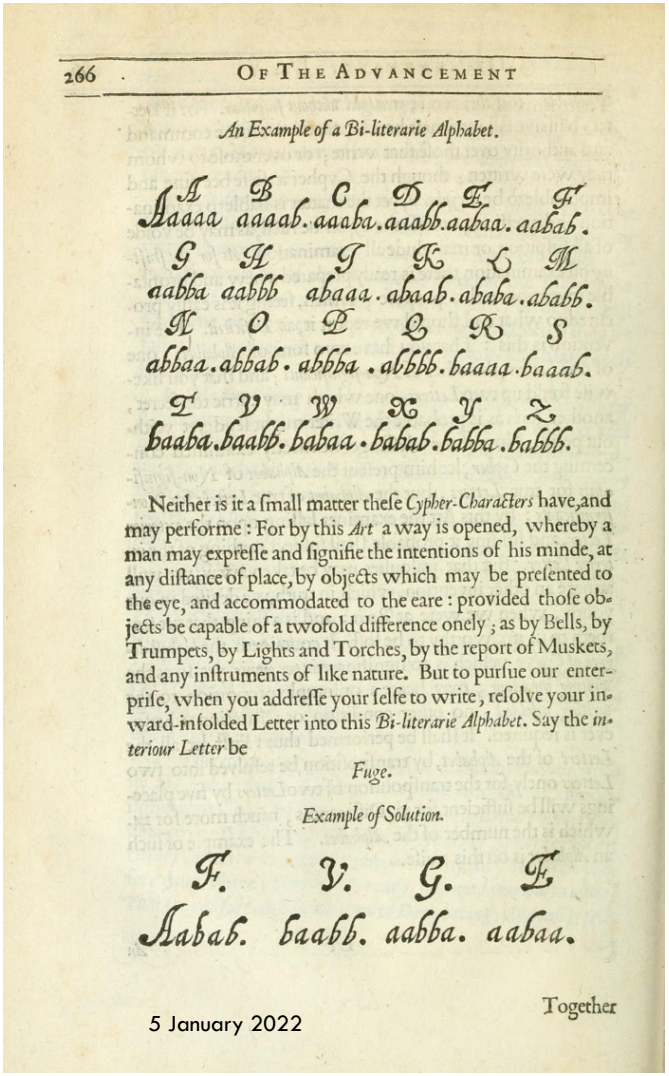
1946: John von Neumann

John von Neumann designed the electronic digital stored program computer as we know it today:



Electronic digital computers use binary

- Engineers assembled electronically-controlled switches into circuits to perform simple calculations
- Positive voltage was designated “1”
- Zero voltage was designated “0”
- 0s and 1s are known as bits: **bi**nary digi**ts**



266 . OF THE ADVANCEMENT

An Example of a Bi-literarie Alphabet.

A aaaaa B aaaab. C aaaba. D aaabb. E aabaa. F aabab.
G aabba H aabbb I abaaa. K abaab. L ababa. M ababb.
N abbaa. O abbab. P abbba. Q abbbb. R baaaa. S baaab.
T baaba. V baabb. W babaa. X babab. Y babba. Z babbb.

Neither is it a ſmall matter theſe *Cypher-Characters* have, and may performe: For by this *Art* a way is opened, whereby a man may expreſſe and ſignifie the intentions of his minde, at any diſtance of place, by objects which may be preſented to the eye, and accommodated to the eare: provided thoſe objects be capable of a twofold difference onely; as by Bells, by Trumpets, by Lights and Torches, by the report of Muskets, and any inſtruments of like nature. But to purſue our enterpriſe, when you addreſſe your ſelfe to write, reſolve your inward-infolded Letter into this *Bi-literarie Alphabet*. Say the *interiour Letter* be

Fuge.

Example of Solution.

F. V. G. E.
aabab. baabb. aabba. aabaa.

Together

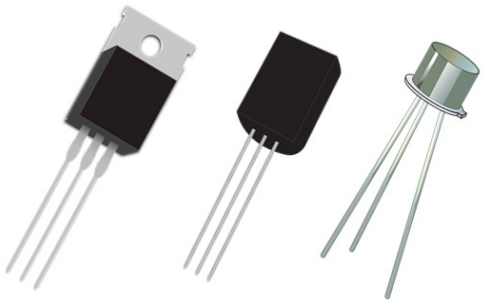
Aside: binary is not new

- The idea of encoding characters is not new
- This is a snippet of Francis Bacon's 5-bit "Bi-literatie Alphabet" (1624)
- Instead of 0s and 1s, Francis used as and bs
  - A = aaaaa aka 00000
  - B = aaaab aka 00001
  - C = aaaba aka 00010
  - D = aaabb aka 00011
  - E = aabaa aka 00100
  - F = aabab and so on.

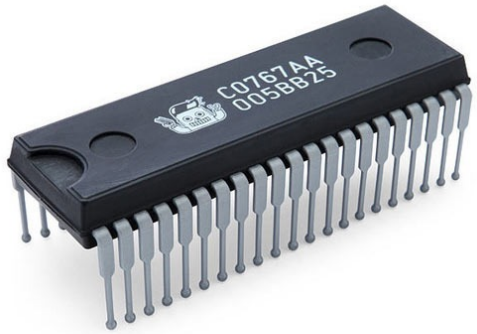
How did we manage complexity?

- Early computers (1930s, 1940s, 1950s) were **huge**
- T**housands of switches** (vacuum tubes) configured for one calculation
- To support different calculations, circuits called processors were created
- **A processor processed (executed) a list of instructions**
- Each instruction performed a **calculation**
- We stored intermediate and final results in memory
- Memory is a circuit that stores 0s and 1s in a series of addressed locations.

Did it end with vacuum tubes?



- **No, that was just the start!**
- 1947: **Transistor** (2nd generation)
- 1958: **Integrated circuit** (3rd generation)

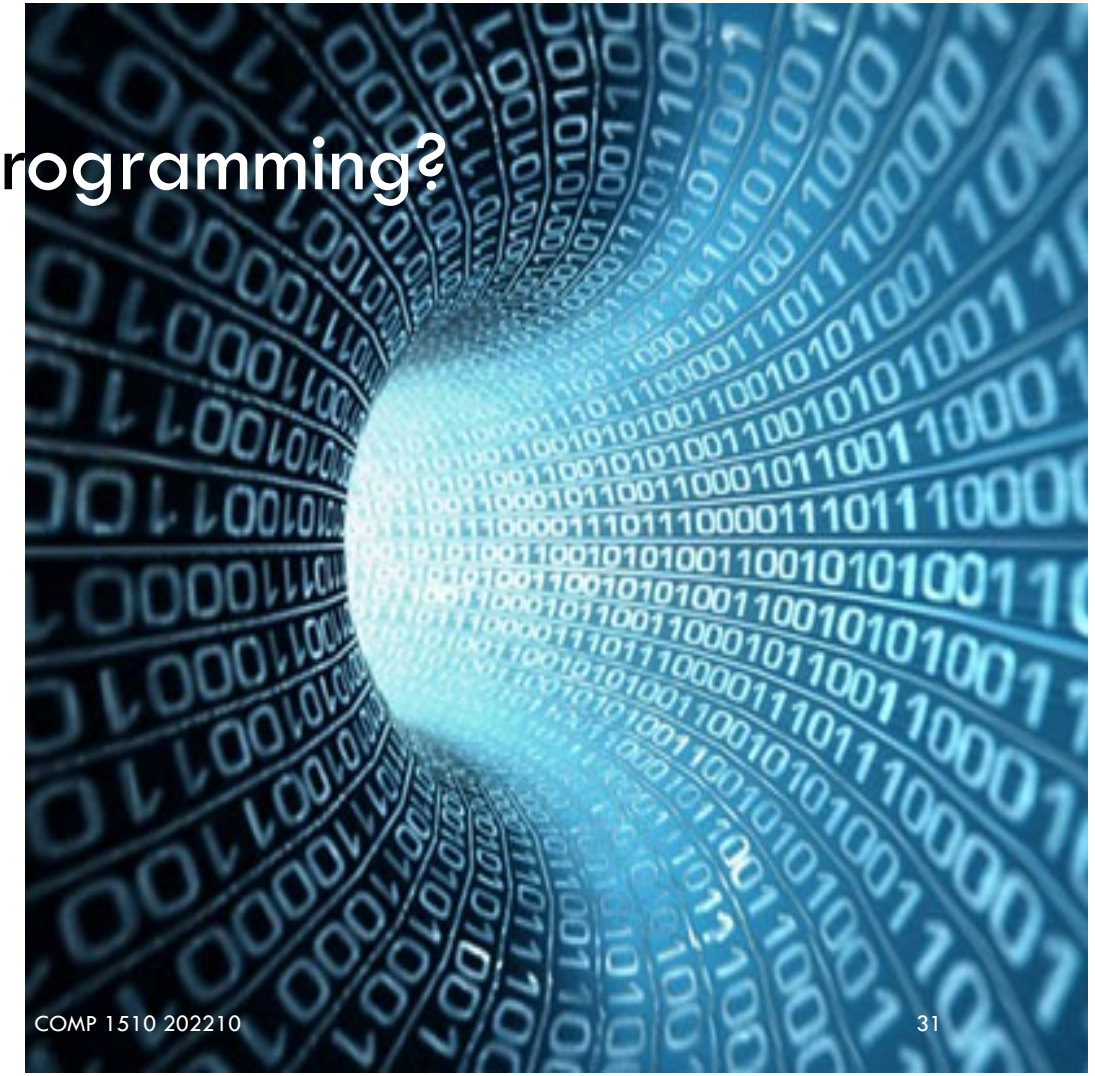


- Trends:
  - Miniaturization
  - Increased speed
  - Increased memory size
  - **Increasingly difficult to program!**

PROGRAMMING LANGUAGE LEVELS

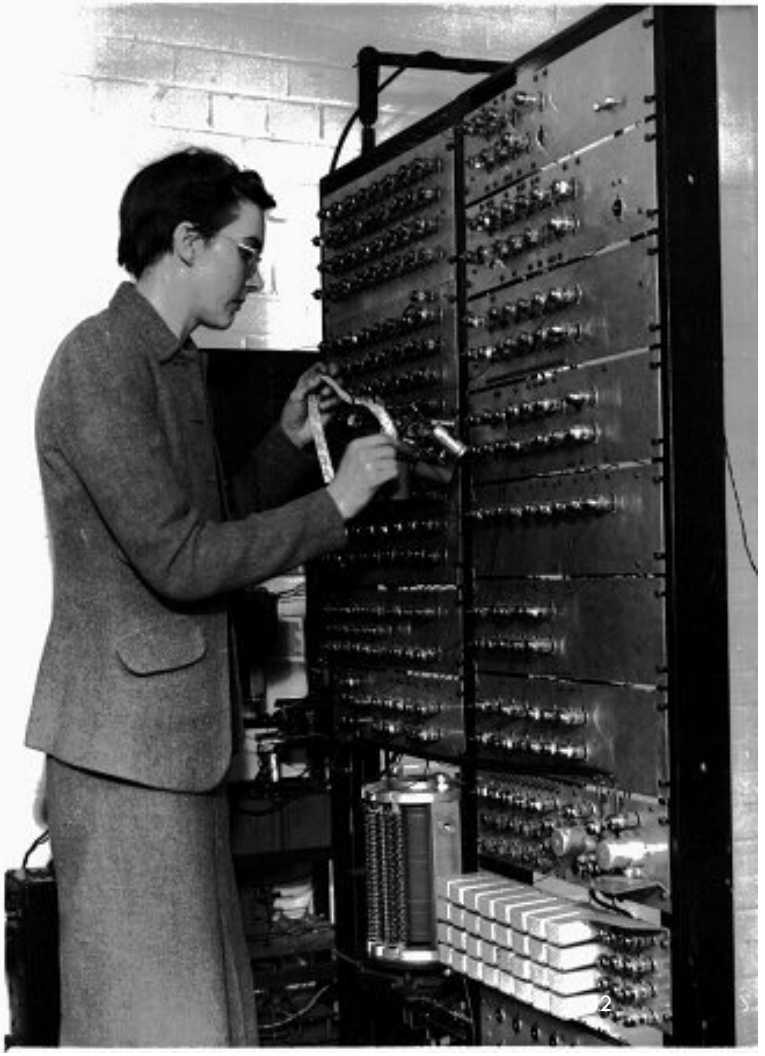
How did we start programming?

- We started by writing our programs in **binary** (0s and 1s)
- (Can you imagine how **tedious** this must have been?)
- Instructions represented as 0s and 1s are known as **machine instructions,** or **machine language**
- Each CPU's instruction set is **unique**



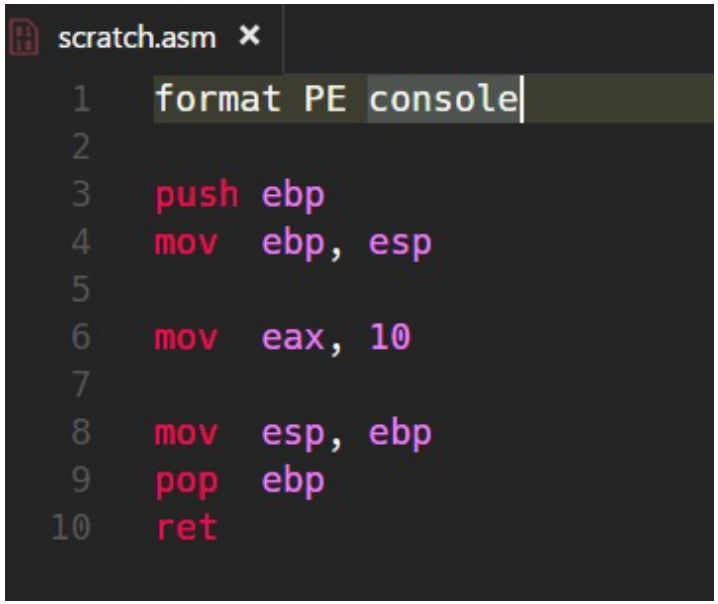
Binary is hard to read

- Machine language is hard for humans to read and write
- **Dr. Kathleen Booth,** a mathematician, developed the first special program called an **assembler** in 1949
- An assembler translates human-readable instructions into binary
- She also wrote the 1st assembly language (collection of instructions that can be processed)
- **No more writing in binary: 10110000 01100001**
- **Now write it in an assembly language like this instead: MOV AL, 61h**
- Dr. Booth emigrated to Canada to teach



Spotlight: x86 assembly language

- Family of backward-compatible languages used to produce code for the ***x86 class*** of processor
- Provides some level of compatibility with April 1972's ***Intel 8008***
- Uses short mnemonics to represents machine language instructions
- You will probably see and maybe use x86 at BCIT.



We can describe levels of languages

1. Level **ONE** is machine language **01010000111010101010101**
   - Each CPU has its own machine language
2. Level **TWO** is called assembly language **MOV EAX, [EBX]**
   - Each CPU has its own assembly language

- These languages are:
  - **old** and **machine dependent**
  - called **low-level languages**
- Low level languages deal with memory, addresses, and simple operations.

High-level languages

- Programmers don't program in machine language (lowest)
- There are still some (very few) specialized uses for assembly…
- Writing complex code takes many instructions
- **High-level languages use abstraction**
- Deal with variables, arrays, objects, complex arithmetic or Boolean expressions, functions, subroutines, loops, locks, etc.
- During the 50s we saw many great high-level languages developed like **FORTRAN**, **COBOL**, **Lisp**

High-level languages are compiled

- Focus is on **usability** and **readability**
- High-level languages use a **compiler**
- A compiler **translates** source code written by a human to a lower-level language that can be processed by a CPU
- US Navy Rear Admiral Grace Hopper, PhD, wrote the first compiler

Kinds of higher level languages

1. **1950s – 1970s 3rd generation languages**
   - Easier for the programmer to write more complex code using abstraction
   - Computer takes care of non-essentials
   - Modern examples include C, Pascal
2. **1970s – 1990s 4th generation languages**
   - Even higher level of abstraction of internal computer hardware details
   - Operate on large collections of information at once instead of bits
   - Examples include MATLAB, C++, Java, and Python
3. **1980s – 1990s 5th generation languages**
   - Accept data and constraints and solve problems without a programmer
   - Examples include Prolog, Mercury.

PROGRAMMING LANGUAGE PARADIGMS

par·a·digm

/ˈperəˌdīm/

Noun

- From the Latin roots “para” (beside) and “deiknunai” (to show)
- a worldview underlying the theories and methodology of a particular scientific subject.
- "the discovery of universal gravitation became the paradigm of successful science"

Programming languages have paradigms

- Describes a way of programming
- Tells us which programming structures to use, and how to use them
- The "Age of Chaos" (not really called that) ended when we all began using ***Structured Programming*** in the 1960s
- Structured programming helps prevent what you will learn to recognize as ***spaghetti code***
- Almost all modern programming is a form of structured programming.

Procedural programming

- You will learn this with me!
- We group our instructions into procedures called functions
- We store the state of the program
- A main procedure (function) starts the program
- It calls procedures (functions)…
- That call procedures (functions)…
- That call procedures (functions)…
- Until the program is done.

My code for a COMP 2510 assignment…

```
117  /* Function to compare elements */
118  int cmp(const void *p, const void *q) {
119
120      char **pp = (char **) p;
121      char **qq = (char **) q;
122      return mystricmp(*pp, *qq);
123  }
124
125  /* Function to compare strings */
126  int mystricmp(const char *s1, const char *s2) {
127
128      /* Variable list */
129      extern int whitespace;
130      extern int lettercase;
131      int leadspaces1 = SET;
132      int leadspaces2 = SET;
133
134      /* Checks for -w flag */
135      if (whitespace == IGNORE) {
136          while ((leadspaces1 == SET) && isspace(*s1)) {
137              s1++;
138          }
139          leadspaces1 = UNSET;
140          while ((leadspaces2 == SET) && isspace(*s2)) {
141              s2++;
142          }
143          leadspaces2 = UNSET;
144      }
145
146      /* Checks for -c flag */
147      if (lettercase == IGNORE) {
148          while (toupper(*s1) == toupper(*s2)) {
149              if (*s1 == 0)
150                  return 0;
151              s1++;
152              s2++;
153          }
```

Functional programming

```haskell
{-
 - Solution to Project Euler problem 205
 - Copyright (c) Project Nayuki. All rights reserved.
 -
 - https://www.nayuki.io/page/project-euler-solutions
 - https://github.com/nayuki/Project-Euler-solutions
 -}

import Data.Ratio ((%))
import Numeric (showFFloat)


main = putStrLn (ans "")
numer = sum [(ninePyramidalPdf !! i) * (sum (take i sixCubicPdf)) | i <- [0 .. ((length ninePyramidalPdf) - 1)]]
denom = (sum ninePyramidalPdf) * (sum sixCubicPdf)
ans = showFFloat (Just 7) (fromRational (numer % denom))


pyramidalDiePdf = [1, 2, 3, 4] :: [Integer]
ninePyramidalPdf = (iterate (convolve pyramidalDiePdf) [1]) !! 9

cubicDiePdf = [0, 1, 1, 1, 1, 1, 1] :: [Integer]
sixCubicPdf = (iterate (convolve cubicDiePdf) [1]) !! 6


convolve :: [Integer] -> [Integer] -> [Integer]
convolve as bs =
    let
        m = length as
        n = length bs
        func i = sum [(as !! j) * (bs !! (i - j)) | j <- [(max 0 (i - (n - 1))) .. (min i (m - 1))]]
    in map func [0 .. (m + n - 2)]
```

* Code from https://github.com/nayuki/Project-Euler-solutions/blob/master/haskell/p205.hs

- Languages like Haskell
- We group our instructions into functions
- The main function begins a program
- It calls functions…
- That call functions…
- That call functions…
- Until the program is done…
- <u>Without ever having stored a state!</u>

Logical programming

- Languages like Prolog
- Based on logic
- Declarative programming language: the program logic is expressed in terms of relations, facts, and rules
- Computations are executed by running queries against the relations

  cat(tom).

  Q cat(tom) A: Yes

  Q cat(X) A: X = tom

```
male(charles).
male(edward).
male(andrew).

female(elizabeth).
female(ann).

child(charles, elizabeth).
child(ann, elizabeth).
child(andrew, elizabeth).
child(edward, elizabeth).

older_than(charles, ann).
older_than(charles, andrew).
older_than(charles, edward).
older_than(ann, andrew).
older_than(ann, edward).
older_than(andrew, edward).

son(X,Y) :- child(X,Y), male(X).

daughter(X,Y) :- child(X,Y), female(X).

successor(X, Y):- child(Y,X).
%successor(X, Y):- son(Y,X).
%successor(X, Y):- daughter(Y,X).

successionList(X, SuccessionList):-
    findall(Y, successor(X, Y), SuccessionList).

precedes(X,Y):- male(X), male(Y), older_than(X,Y).
precedes(X,Y):- male(X), female(Y), Y\=elizabeth.
precedes(X,Y):- female(X), female(Y), older_than(X,Y).

%precedes(X,Y):- older_than(X,Y).

%% Sorting algorithm
succession_sort([A|B], Sorted) :- succession_sort(B, Sorted
succession_sort([], []).

insert(A, [B|C], [B|D]) :- not(precedes(A,B)), !, insert(A,
insert(A, C, [A|C]).



successionListIndependent(X, SuccessionList):-
    findall(Y, child(Y,X), Children),
    succession_sort(Children, SuccessionList).
```

Object oriented programming

- Languages like C++, C#, Java, and ***Python***
- HOT, HOT, HOT
- Long history (1960s), ubiquitous in the industry
- Organize our code to represent objects with behaviours
- A program is a collection of interacting objects
- Focus is on encapsulation, **inheritance***, **polymorphism***.

* See COMP 2522

What will we be using

- That's the million-dollar question
- Historically, COMP 1510 used Java (and the BBY campus still uses it!)
- Java is a great language and I love it and use it in COMP 2522 and in real life
- But in 2018 I started using Python for COMP 1510
- I really like it!
- And so will you. I promise!

INTRODUCING PYTHON

I ❤️ Python

- Modern and popular
- **Easy** to learn
- Lots of **support** on the cloud
- Large and comprehensive **standard library**
- Supports multiple programming paradigms like **OOP**, functional, procedural
- It's an **interpreted** (not compiled*) language
- You can apply ***everything*** you learn in this course to other languages

* Almost true…

Which version?

- There are 2 versions of Python: Python 2 and Python 3
- We will use Python 3
- Now is a good time to bookmark some online resources:
    1. https://www.bcit.ca/outlines/20221086126/
    2. http://python.org/
    3. https://docs.python.org/3/
    4. https://docs.python.org/3/library/index.html

Installing Python

- Windows use this link: https://www.python.org/downloads/
- macOS may choose to install and use Homebrew (https://brew.sh) a very good macOS "Package Manager"
  1. First ensure XCode is installed and up to date
  2. Open a Terminal window
  3. Install Homebrew by copying and pasting this into the window:
     /bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
  4. When it is done, enter the command:
     brew install python3

Python demo

- I start by opening a Terminal (Command Prompt) window
- Type **python --version** to see if it's installed and, if so, whether it's Python 3.x (that's a double dash)
- If it's not what we want, type **python3 --version** (that's a double dash)
- Use the command **python3** to open the Python 3 interpreter and begin the read-evaluate-print loop (REPL)
- This opens a prompt
- We can also call this a "*command line session with an interactive interpreter*"
- We can leave it by typing **quit( )** (no spaces)

Python keywords

`False`	`await`	`else`	`import`	`pass`
`True`	`break`	`except`	`in`	`raise`
`None`	`class`	`finally`	`is`	`return`
`and`	`continue`	`for`	`lambda`	`try`
`as`	`def`	`from`	`nonlocal`	`while`
`assert`	`del`	`global`	`not`	`with`
`async`	`elif`	`if`	`or`	`yield`

ZEN OF PYTHON

Try this:

1. Open your Python 3 interpreter
2. Enter this command:

```
import this
```

Zen of Python

- These software principles guided the creation of Python
- Read this every evening before you go to sleep
- Read this every morning as you prepare for your day

CLI (COMMAND LINE INTERFACE)

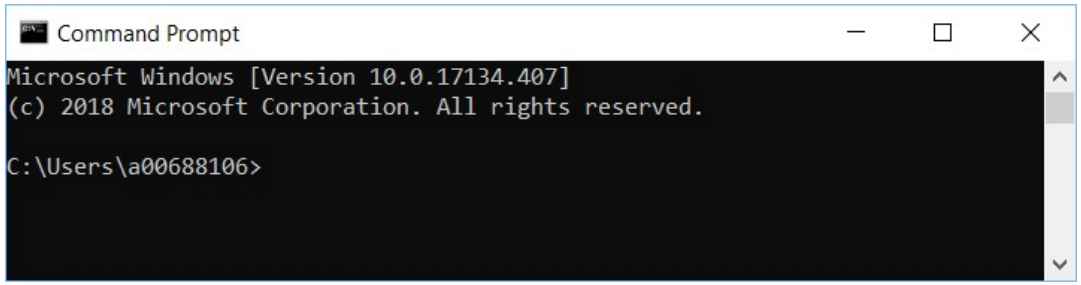
The “Command Line” (Interface) aka CLI

- Windows calls it the **Command Prompt**
- macOS calls it the **Terminal**
- Unix and its relatives like Linux call it the **Terminal,** too
- It is a text-based interface to our operating system
- We interact with the operating system using specific words called commands
- Programmers spend a lot of time on the command line
- Many programmers think it is a great virtue to do everything with the keyboard and nothing with a mouse
- A skilled programmer working the command line is a wondrous sight 💯

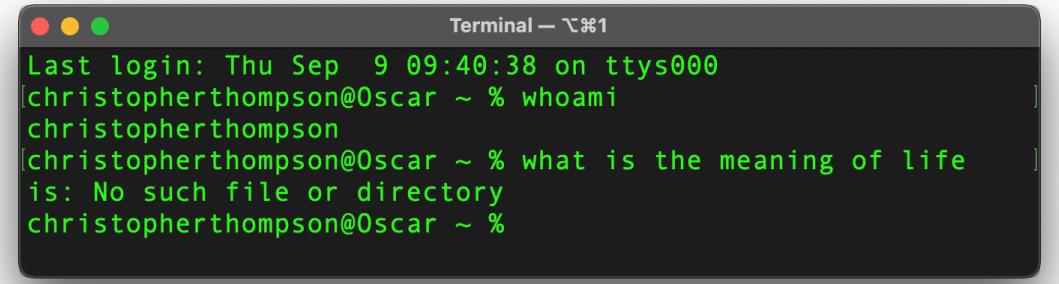
Command line basics

- When we open the command line we usually begin at our home directory:

Windows



macOS



Command line basics

- Clear the terminal screen:
  - Windows: cls
  - macOS and Unix: clear
- Print the name of the current directory aka folder:
  - Windows: cd (stands for current directory)
  - macOS and Unix: pwd (stands for print working directory)
- Make a new directory:
  - Windows: mkdir name_of_directory
  - macOS and Unit: mkdir name_of_directory
- Print a list of environment variables:
  - Windows: set
  - macOS and Unix: env

Another slide of command line basics

- Print contents of current directory
  - Windows: dir
  - macOS and Unix: ls
- Rename a file or directory:
  - Windows: rename old_name new_name
  - macOS and Unix: mv old_name new_name
- Enter directory named alpha:
  - Windows: cd alpha
  - macOS and Unix: cd alpha
- Move “up” one directory (move to the directory that contains the current directory):
  - Windows: cd ..
  - macOS and Unix: cd ..

Final slide of command line basics

- Delete a file:
  - Windows: del file_name
  - macOS and Unix: rm file_name
- Current directory:
  - Windows and macOS and Unix: .
- Parent directory:
  - Windows and macOS and Unix: ..

Examples:

- cd ../.. to move up two levels
- C:\user\docs\Letter.txt (we can change drives by typing DriveLetter followed by :)
- cd ~\Desktop (macOS only where ~ is our home directory)

But Chris, what's a path?

- A path specifies a unique location in a file system
- It is expressed as a **hierarchy of directories** separated by a delimiting character
- For example on my Windows laptop:
  1. My system partition is called C: (this is my root directory)
  2. C: contains a directory called Users
  3. Users contains a directory called a00688106
- The path to the a00688106 directory is C:\Users\a00688106

Is it the same using macOS or Unix?

- Almost
- The delimiting character can be different
- In Windows it is a \ or a /
- In macOS and Unix it is always /
- On my Mac:
  - I have a hard drive whose root directory is denoted with the /
  - The root directory contains a directory called Users
  - Users contains a directory called christopherthompson
- The path to the christopherthompson directory is /Users/christopherthompson

What else should we know about?

Absolute path:

- Points to the same location in the file system regardless of working directory
- Always begins with the root directory and works "inward"
- It's how to get there from the root of the tree

```
> pwd
/home/chris
> cd /home/chris/exams
> pwd
/home/chris/exams
```

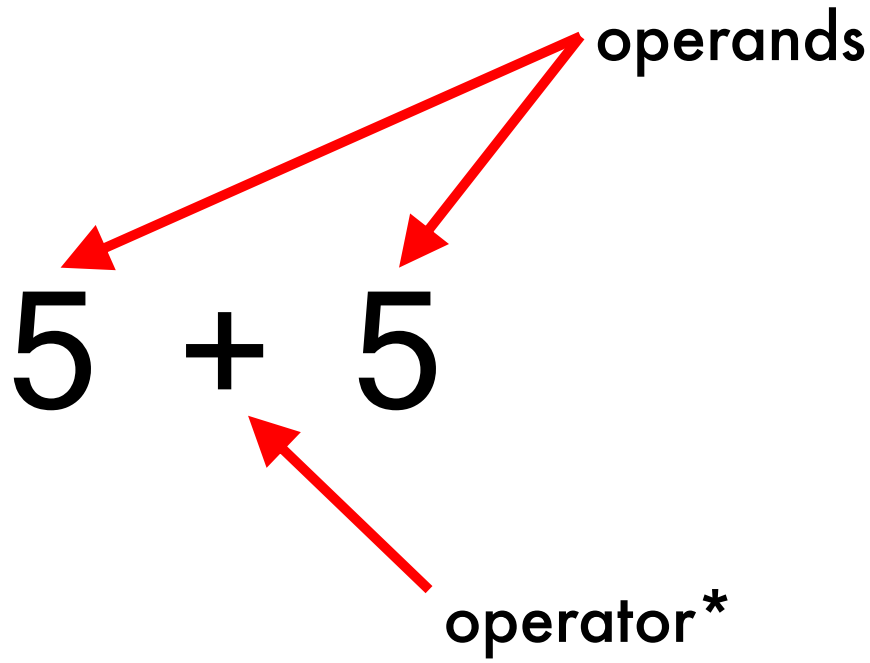
Relative path:

- Starts from the current working directory
- Avoids the need to provide the entire path
- It's how to get there from where you are right now

```
> pwd
/home/chris
> cd exams
> pwd
/home/chris/exams
```

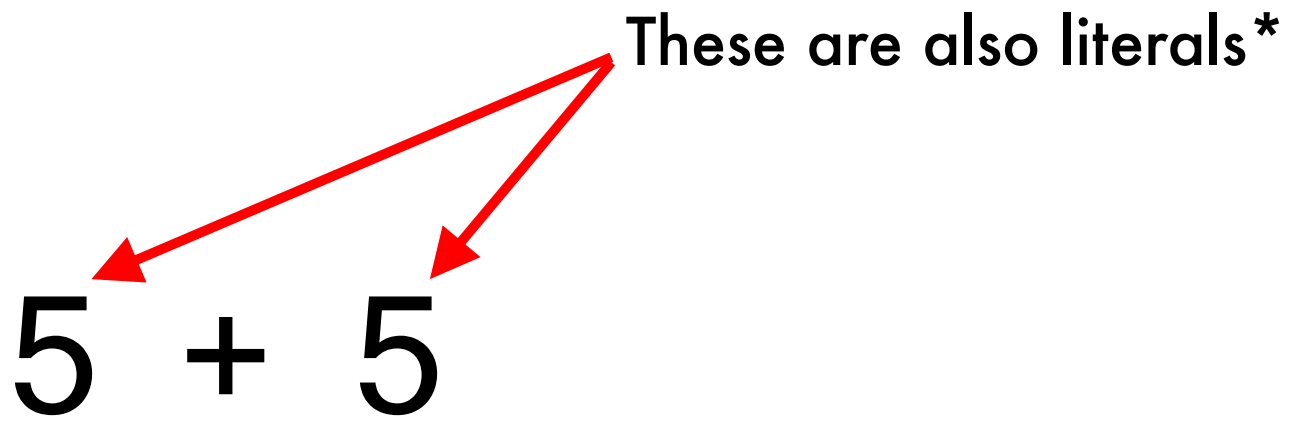
OPERATORS, OPERANDS, AND EXPRESSIONS

Operators and operands



* Actually + is a “binary infix operator”

Literals are raw values



* Actually these are called “numeric literals”. We also use string literals like “Hello”, the Boolean literals True and False, and collection literals for things like lists, dictionaries, sets, and tuples.

What kind of arithmetic operators are there?

Operator	Name	Example
+	Addition	1 + 2.0
-	Subtraction	3.14 – 1.4142
*	Multiplication	5 * 5
//	Integer Division	4 // 2 (try 3 // 0)
/	Division	6.0 / 4
%	Remainder/Modulo	10 % 4
**	Exponent/Power	4 ** 4

What else is there?

Operator	Name	Example	
and	Boolean and	4 < 5 and 9 < 10	Logical operators
or	Boolean or	6 < 5 or 9 < 10	
not	Boolean not	not True	
is	Identity	"better" is "bad"	Identity operators
is not	Identity	2 is not 2.0	
in	Membership	"C" in "Chris"	Membership operators
not in	Membership	4 not in [1, 2, 3]	

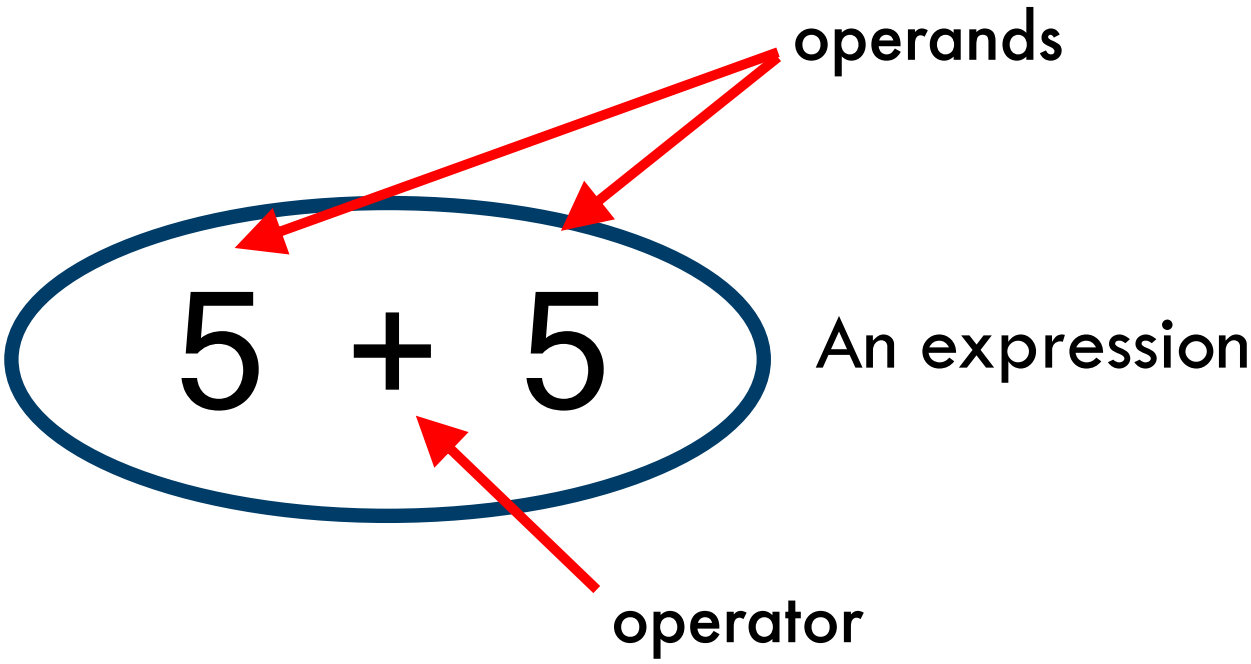
OMG there's more?!

Operator	Name	Example
<	Strictly less than	5 < 4
>	Strictly greater than	"z" > "Z"
<=	Less than or equal to	5 <= 7
>=	Greater than or equal to	"better" >= "bad"
==	Equal to	2 == 2.0
!=	Not equal to	3.14 != 3
<<	Bitwise left shift	2 << 5
>>	Bitwise right shift	64 >> 5
&	Bitwise and	2 & 3
|	Bitwise or	2 | 4
^	Bitwise xor	7 ^ 2
~	Bitwise not	~2
:=	Walrus operator	while (current := input("Write something: ")) != "quit":

Comparison operators

Bitwise operators

Operators and operands form expressions



What's an expression?

- In math (and computing) it is a **sequence** of operators and their operands
- The sequence specifies a **computation**
- Evaluating an expression produces a **result (value)**
- The expression 2 + 2 generates the result 4
- Evaluating an expression may generate a **side effect**, i.e., evaluating print('BCIT is awesome') sends a string of characters to standard output (the console).

Expression: operators and operands

A value all by itself is a simple expression

OPERATOR PRECEDENCE

Precedence so far (and some sneak peaks!)

Operator	Description	Explanation
( )	Items within parentheses are evaluated first	In (a * (b + c)) - d, the + is evaluated first, then *, then -.
** * // / % + -	Arithmetic operators (using their precedence rules; recall ***PEDMAS*** from high school)	z - 45 * y < 53 evaluates * first, then -, then <.
< <= > >= == !=	Relational, (in)equality, and membership operators	x < 2 or x >= 10 is evaluated as (x < 2) or (x >= 10) because < and >= have precedence over or.
not	Logical NOT	not x or y is evaluated as (not x) or y
and	Logical AND	x == 5 or y == 10 and z != 10 is evaluated as (x == 5) or ((y == 10) and (z != 10)) because and has precedence over or.
or	Logical OR	x == 7 or x < 2 is evaluated as (x == 7) or (x < 2) because < and == have precedence over or

https://docs.python.org/3/reference/expressions.html#operator-precedence

COMMANDS, VARIABLES, AND ASSIGNMENT

Commands aka statements

- **Code** is the textual representation of a program
- Programming is often called **coding***
- A single row of text is called a **line of code**
- A line of code usually contains a **command** or a definition
- A command is often called a **statement**
- A command (statement) is a program instruction (an order that we "execute")
- It tells the Python interpreter to do something, like print or add.

* But some developers think "coder" is a pejorative and prefer the term "programmer"

A Python program

- Sometimes called a **script**
- A **sequence of executable statements**
- These statements are evaluated and the commands inside them are executed **one at a time in order** by the Python **interpreter**
- We say that execution takes place inside the Python **shell**
- Typically, a new shell is created whenever execution of a program begins
- The shell is how we interact with the interpreter.

Sample statements (let's print stuff)

```
print('Able was I ere I saw Elba')
print('Out darn spot, out!')
print('Season 8 was hot garbage 😤')
print(4 + 8)
print(3 * 2.5 ** 2 + 2.5 * 2 + 4)
```

Variables

A variable provides a way to associate a name with a value

```
pi = 3.1415926535
colour = “red”
```

The first line of code binds the name pi to a location in memory that stores the floating point value 3.1415926535

The second line of code binds the name colour to a location in memory that stores the ordered sequence of three Unicode characters r, e, and d.

* We always wrap strings in double quotes. Why?

Assignment

- A variable is just a name
- We must assign a value to a variable to make it useful
- An **assignment statement** associates some value on the right of an equal sign with the variable on the left of the equal sign
- **We do this with the equal sign =**
- The left side must be a variable
- The right side can be a value or an expression

Hint: if you can print it, or assign it to a variable, it's an expression, otherwise it is a statement.

Sample assignment statements

```
user_name = “Chris”
password_is_correct = False
result = 3 * 2.5 ** 2 + 2.5 * 2 + 4
pi = 3.14
radius = 6.0
circumference = 2 * pi * radius
minimum_comfortable_temperature_celsius = 10.0
maximum_comfortable_temperature_celsius = 24.0
tolerance = maximum_comfortable_temperature_celsius -
            minimum_room_temperature_celsius
```

Variable names

- Each variable needs a meaningful name
- We call this its **identifier**
- Identifiers are the names for things in Python
- An identifier can be composed of:
  1. **Letters** (a-z, A-Z)
  2. **Underscores** (_)
  3. **Digits** (0-9)
- An identifier must start with a letter or underscore
- Python is case sensitive, i.e., Python and python are different

Identifiers must describe purpose

- If a variable in a game stores an NPC's age, a variable name like "age_in_years" is better than a name like "a" or even "age"
- Avoid single character variable identifiers. Always. They make me cry.
- Compare:

```
a = 3.14159          pi = 3.14159
b = 20.18            radius = 20.18
c = a * (b**2)       area = pi * (radius**2)
```

- Abbreviations should only be used when widely understood, like "app" or "faq"

Aside: we have different cases in English

1. ThisIs**Camel**Case
2. This Is **Title** Case
3. THIS IS **UPPER** CASE
4. this is **lower** case
5. in_python_we_do_this*

* Snake case. Because Python, get it? 😃 🐍

That's it for week 01!

Any questions?