# What is programming?

# An example...

You've probably programmed before. When you enter 1+1 on a calculator, you are commanding it to calculate 1+1 for you.

Now, what if you want your calculator to calculate lots of equations for you over and over in the future? Well, you write those equations out on a text file and feed this text file into your calculator for it to calculate the equations in the text file for you. This way, you can re-feed this text file any time you want the calculator to perform the same calculations in the future i.e. you:

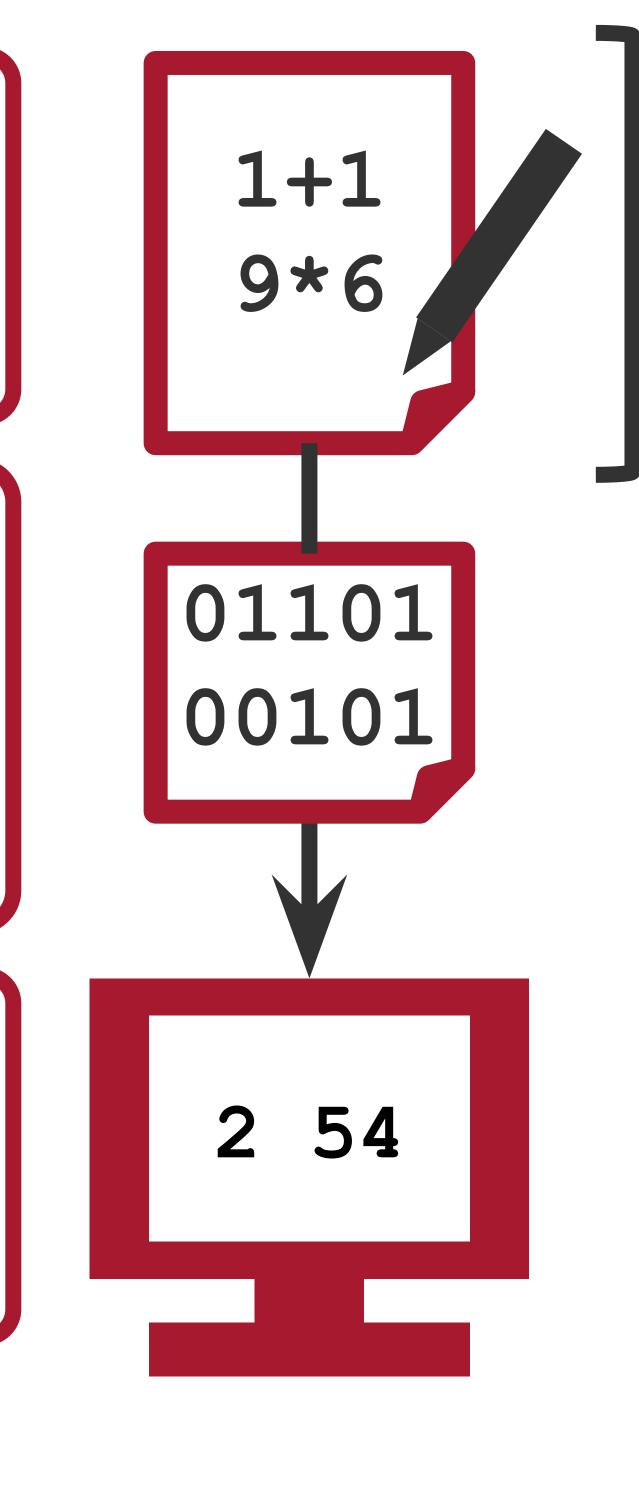
- "programmed" (wrote) a set of equations or instructions into a "program" (a text file) so that
- your "computer" (calculator) could "execute" (do the calculations requested by) your program.

# of a programming workflow

A computer is just a super powerful calculator... that can only read and execute programs written in binary i.e. Os and 1s. We call these "binary" program files, "machine executable" files.

Binary program code is almost impossible for us to read or write, so we came up with this workflow:

- 1. program: you write a program in a human-readable "programming language" to a program text file.
- 2. compile: you get a "compiler" (translator) to "compile" (translate) your program text file into a machine read-able binary file.
- 3. execute: you let the computer execute the instructions in this binary.

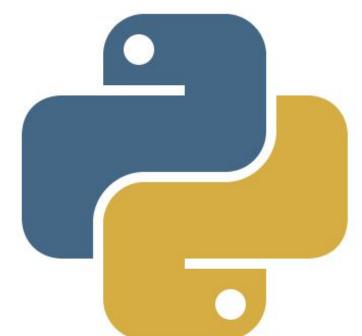


# C++ and Python are two of the most popular programming languages. — J



C++ is more difficult to write, but it is extremely efficient and powerful.

Operating systems (OS) and lower-level software are primarily written in C++.



Python is much easier to write and is very popular, but it gives you less control. Python is used for high-level applications such as software development and artificial intelligence.

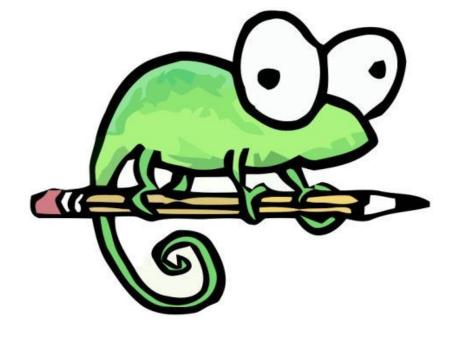
# Text editors and IDEs

Let's start with getting some tools to complete step 1: writing a human-readable program.

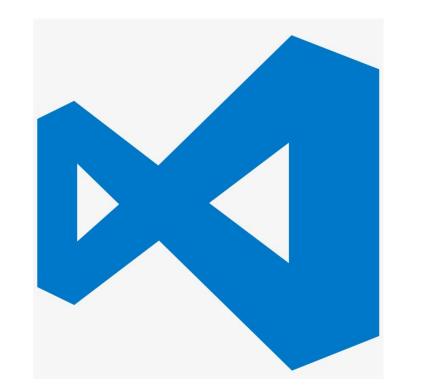
Your program code is essentially a text file, so you can edit it in any text editor. There are two types of text editing software you can use to write your program:

Text editors are lightweight text editors that may have a few additional functionalities such as syntax (grammar) highlighting.

IDEs (integrated development environment) are heavyweight text editors plus all the other tools you will need during the programming process. These tools are usually language-specific.

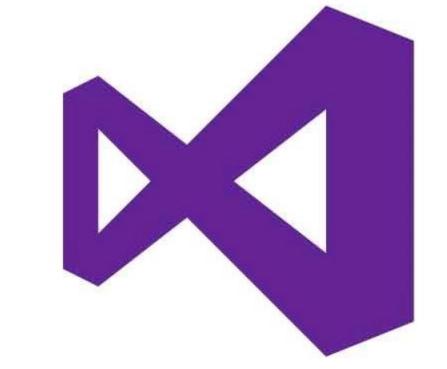




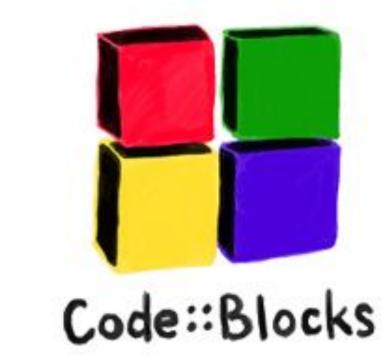




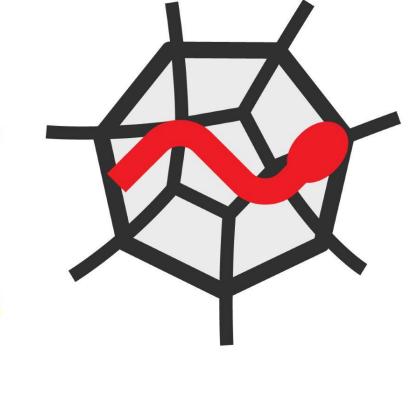












Notpad++ simple, fast

Sublime text simple, fast

VS Code extendable

<u>Atom</u>

<u>Vim</u> extendable hard, flexible

<u>Visual Studio</u> generic

<u>Eclipse</u> generic

Code::blocks C++

<u>Pycharm</u> Python

<u>Spyder</u> Python

# Set up your computer for programming with a **terminal** (C++ and Python)

# Prepping your terminal

The easiest way to set up for programming is via a Unix terminal.

Windows: On Windows 10+, your OS comes with a Windows subsystem for Linux (WSL) which allows you to interface with your OS via a Unix terminal.

Let's enable WSL and install a Linux distribution if you don't have it installed already:

- open PowerShell as administrator:
 start > search for "PowerShell" >
 right-click "Windows PowerShell" and

select "Run as administrator".

- paste the following command into the PowerShell and press the **enter** key:

Enable-WindowsOptionalFeature
-Online -FeatureName
Microsoft-Windows-Subsystem-Linux

- restart your computer on prompt.
- now that you have WSL enabled, let's install the Ubuntu distribution <a href="here">here</a>!
- after you've installed Ubuntu, open it up! If this is the first time you're opening it up, it should say it's installing. After that is finished, give yourself a username and password.

Now that you have a terminal (like the one you would have on Ubuntu), you will need to **install**:

- a C++ compiler g++ by installing the GNU compiler tools and the GDB debugger (sudo means to run as admin, whereis verifies successful installation),

\$ sudo apt-get update
\$ sudo apt-get install
build-essential gdb
\$ whereis g++

- and a Python 3 interpretor.

\$ sudo apt update && upgrade
\$ sudo apt install python3
python3-pip ipython3

Now you're all set up to use your terminal in Windows to work with C++ and Python:)!

Linux & macOS: Python and C++ are pre-installed and set up on your machine! It's just a matter of opening up the terminal and using them.

For Linux computers, this guide assumes you are using a Gnome-based desktop environment (e.g. Ubuntu) that interfaces between you and Linux.

Let's open up your terminal:

application menu / launchpad >
search for "terminal" and select it.

# Using your terminal

The terminal is an interface where you can type commands for your OS to execute. *Think of your terminal* as a *dumbed-down Siri* who only takes super specific typed out commands; but as long as you know what the commands are, your terminal can do almost anything! Including executing your C++ or Python program.

Navigating your files from your terminal: before programming, we need to learn how we can navigate our files. Here's a few basic commands:

- \$ ls: list the contents in your current directory.
- \$ cd [directory]: "change directory" i.e. go to the [directory] or the folder (that you saw when you ls-ed);
  - \$ cd /: root directory on Linux & macOS.
  - \$ cd /mnt/c: the C drive on Windows.
  - \$ cd .: the parent directory.
- \$ mkdir [directory]: make a new directory.
- \$ rm [file] / \$ rm -rf [directory]: remove (delete) a file / directory.
- \$ cp [file1] [file2]/
  \$ cp -r [directory1] [directory2]:
  copy and paste [file1]/[directory1] as
  [file2]/[directory2].
- \$ cat [file]: show contents of a [file].

**Pro tip**: press **ctrl-c** inside the terminal to terminate or stop any command.

# 1. Program Syntax

# **C++**

Declares a variable num

- of data type int (integer),
- value of 0, and
- creates a **pointer** (address) to the space in memory where C++ stores its value.

All statements end with;

std::cin takes a user inputfrom the terminal and assignsit to num in replacement ofits current value 0.

std::cout prints to standard
output << "You entered"
and sqrt(number).</pre>

int is the output return type
of main. In this case, it was 0.

return terminates main and ends the program.

#### main.cpp

```
// author: alice yue
// date: 2021-05-21
```

// description: does math!

#include <iostream>
#include <cmath>

return 0;

num << " is " << sqrt(num);

std::cout << "The sqrt of " <<

int main() {...} defines
 a special function called
 main.

Everything after // are

not ran, they're comments!

Use them for explanations.

#include's iostream, cmath

which contains cout/cin and

sqrt, which we use below.

C++ runs a program by by executing everything in the main() body {...}.

Put this in all your programs so that your program runs.

# Python

Declares a variable num.

#### input():

- prints to standard output "Enter an integer: ",
- returns user input, and
- assigns (with operator = ) the input to num.

print() prints to standard
output "You entered " and
sqrt(num).

#### main.py

```
# author: alice yue
# date: 2021-05-21
# description: does math!
```

import math

```
num = input("Enter an integer: ")
print("The sqrt of ", num, " is ",
    math.sqrt(num))
```

Everything after # are not ran, they're comments!
Use them for explanations.

import's package math which
 contains sqrt used below.

# 2. Compile & 3. Execute

C++ Python

# Terminal

g++ "compiles" main.cpp into program.

\$ g++ -o program main.cpp

\$./program

execute program

Enter an integer: 4

The sqrt of 4 is 2

#### Terminal

python "interprets" (compile + execute) main.py.

\$ python main.py

Enter an integer: 4

The sqrt of 4 is 2

(++

# Program Syntax

# Python

#### Variables

A variable is a defined value that holds data.

```
int i = 0;
```

Each variable can be defined by its:

- 1. data type; the data type determines how much space in memory C should allocate for your variable. You can also get the size of a variable via sizeof(i); Common data types: int (e.g. 0), char (e.g. 'a'), float (e.g. 1.2).
- 2. variable name.
- 3. **value**; a variable is "**passed**" as its value by default i.e. the variable name == its value.
- 4. **pointer**; an address to the space C++ allocated for your variable value in the memory.

#### Array; vector

An array is a static (meaning it cannot change in length) list of values of the same data type; unlike variables, an array is "passed" as its pointer by default i.e. the variable name of an array is synonymous to its pointer.

The index of a list starts at 0 and can be

```
int ia[3] = {6,3,1};
ia[0]; // 6
ia[1]; // 3
ia[2]; // 1
```

A vector is an array with inbuilt functions that

```
#include <vector>
...
std::vector <int> iv(3); // declare
```

Strings are just special char arrays that you can

```
std::string str = "Hello World!";
```

The notion of value vs pointer is what confuses most people, make sure you know it by heart:

Declaration	Value	Pointer
int i	i	& i
int* i	* i	i
int ia[3]	*ia	ia

**Pro tip:** you can pull up documentation about a certain, e.g. function, in the Python interpreter using **help** in the interpretor which you can open up in your terminal:

```
$ python
>> help(str)
```

#### Variables

A variable is a defined value that holds data.

```
int i = 0;
```

Each variable can be defined by its:

- 1. variable name.
- 2. value

You can convert a variable to a certain data type using: str(i), int(i), float(i), bool(i).

In Python, each data type has a specific set of **methods**, functions that you can apply to variables of that data type. Methods for strings str include:

# List; dictionary

A **list** is a dynamic group of values not necessarily of the same data type.

The index of a list starts at 0 and can be accessed with [...].

```
11 = ['Hello', 1]
11[0] # 'Hello'
11[1] # 1
```

Python dictionaries are another way to store a list of values, but for each value, its index can be given a unique key.

```
dict = {
    "brand": "Honda",
    "colours": ["red", "blue"],
    "year": 2020
}
dict["brand"] # "Honda"
```

### Arithmetics

Arithmetic operators help manipulate numerics:

```
operators:
```

```
o +, -, *, /, % (modulus/remainder);
e.g. int i=4+5;
```

shorthands:

```
++, --; e.g. i++; is the same as i=i+1;
+=, -=, *=, /=; e.g. i+=1; is the same as i++;
```

#### if/else statements

if statements help add decision-making to your program; its syntax works like this: if (some statement) is true, execute the code in {...}. You can also optionally add else if and else statements if you want to embed more decision-making options.

```
if (i > 2) {
    // code runs if (...) is true
} else if (i == 2) {
    // code runs if (...) is true
} else { // i < 2
    // code runs otherwise
}</pre>
```

The following are infix operators you can use in conditional statements (...) to help you with your decision making:

name	syntax
equal to	
not equal to	!=
less than	<
greater than	>
less than or equal to	<=
greater than or equal to	>=
not	

#### Arithmetics

Arithmetic operators help manipulate numerics:

- operators:
  - +, -, \*, /, % (modulus/remainder);
    e.g. i=4+5;
- shorthands:

```
○ +=, -=, *=, /=; e.g. i+=1; is the same as i++;
```

#### if/else statements

if statements help add decision-making to your program; its syntax works like this: if (some statement) is true, execute the code indented below. You can also optionally add elif and else statements if you want to embed more decision-making options.

**Hint**: indentation is super important in Python, it does not use braces!

```
if i > 2:
    # code runs if statement is true
elif i == 2:
    # code runs if statement is true
else: # i < 2
    # code runs otherwise</pre>
```

The following are logical operators you can use in conditional statements to help you with your decision making:

name	syntax
equal to	==
not equal to	!=
less than	<
greater than	>
less than or equal to	<=
greater than or equal to	>=
not	

## Loops

If you have a list of values you want to loop through, repeating the same piece of code for each value, you can repeat that piece of code over and over again (right; BAD) OR use any one of the following loops (left).

```
int i = 0;
std::cout << i;

i = 1;
std::cout << i;

i = 2;
std::cout << i;

i = 3;
std::cout << i;</pre>
```

while loop: while my condition (...) is true, execute the code in {...}.

```
int i = 0;
while (i < 5) {
    std::cout << i++;
}</pre>
```

do/while loop: do execute
{ . . . }; while my condition
( . . . ) is true, repeat; this
guarantees that { . . . }
executes at least once.

```
int i = 0;
do {
    std::cout << i++;
} while (i < 5);</pre>
```

for loop: (initialize
variable i to 0; for as long
as i < 5, execute the code
in { . . . } and i++ at the
end of each execution).</pre>

```
for (int i=0; i<5; i++) {
    std::cout << i;
}</pre>
```

#### Functions

When your code gets long, it's useful to encapsulate repeating code into a function:

```
// function definition
int mean(int a, int b) {
   int mn = (a + b) / 2;
   return mn;
}

// calling (using) the function
void main () {
   int mn1 = mean(5, 6);
}
```

A function definition includes:

- a return type: int
- function name: mean
- input arguments: (int a, int b)
- function body: { . . . }

#### Loops

If you have a list of values you want to loop through, repeating the same piece of code for each value, you can repeat that piece of code over and over again (left; BAD) OR use any one of the following loops (right).

```
while loop: while my condition (...) is true, execute the code in {...}.
```

```
i = 0;
while i < 5:
    print(i)
    i += 1</pre>
```

```
i = 0
print(i)

i = 1
print(i)

i = 2
print(i)

i = 3
print(i)

i = 4
print(i)
```

for loop: for each element in some list, run the indented code.

```
fruits = ['apple', 'banana', 'orange']
for fruit in fruits:
    print(fruit)
```

#### Functions

When your code gets long, it's useful to encapsulate repeating code into a function:

```
def mean(a, b):
    mn = (a + b) / 2
    return mn

mn1 = mean(5, 6)
```

A function definition includes:

- a keyword: def
- function name: mean
- input arguments: (int a, int b)
- the function body is indented def

# SFU programming resources

## Remote accessing SFU computers

If you ever need access to computers or software at SFU, you can!

- 1. Click <a href="here">here</a> to set up MFA (multi-factor authentication), you will need to do this before you start remote accessing any SFU computers.
- 2. Click <a href="here">here</a> for instructions on how to access the Burnaby or Surrey library lab computers and software from off-campus.

If you are a CMPT student, you will also have access to CSIL (computing science instructional labs).

- 1. Click <a href="here">here</a> to set up MFA (multi-factor authentication), you will need to do this before you start remote accessing any SFU computers.
- 2. Click <a href="here">here</a> for instructions on how to access CSIL via SSH in your terminal.

#### Need software?

Check out the software made available to you

- as SFU students <u>here</u> and
- as SFU CMPT students <a href="here">here</a>!